```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import java.util.TreeSet;
import java.util.LinkedList;
import java.util.HashMap;
import java.util.Iterator;
import letter.Letter;
import equation.GroupEquation;
import equation.QuadraticSystem;

/**
 *
 * @author grouptheory
 */
public abstract class AbstractCancellationDiagramAnalysis
    implements ICancellationDiagramAnalysis {

    private GroupEquation _problem;
    private QuadraticSystem _qs;
    private HashMap _equiv;
    private GroupEquation _problemQuadratic;
    private LinkedList _decorators;

    protected AbstractCancellationDiagramAnalysis(GroupEquation problem, QuadraticSystem qs) {
        _problem = problem;
        _qs = qs;

        _problemQuadratic = qs.getEquation();
        _equiv = new HashMap();
        _equiv.putAll(qs.getEquivalences());

        _decorators = new LinkedList();
    }

    public final void addDecorator(ICancellationDiagramAnalysisDecorator dec) {
        _decorators.addLast(dec);
    }

    public final Iterator iteratorDecorators() {
        return _decorators.iterator();
    }
```

```
    public final GroupEquation getProblem() {
        return _problem;
    }

    public final QuadraticSystem getQuadraticSystem() {
        return _qs;
    }

    public final GroupEquation getQuadraticEquation() {
        return _problemQuadratic;
    }

    public final HashMap getEquivalences() {
        return _equiv;
    }

    public String toString() {
        String s = "";

        s += ("Quadratic Equation: "+_problemQuadratic+" = 1\n");

        s+="\n\nVariable equivalences:\n\n";
        for (Iterator it=_equiv.keySet().iterator(); it.hasNext();) {
            Letter let = (Letter)it.next();
            s += (let.toString() + "="+ ((Letter)_equiv.get(let)).toString() + "; \n");
        }

        return s;
    }

    static class FilteredIterator implements Iterator {
        private Iterator _it;
        private DiagramTreeNode _nextDTN;

        FilteredIterator(Iterator it) {
            _it = it;
            _nextDTN = nextDTN();
        }

        public boolean hasNext() {
            return (_nextDTN!=null);
        }

        public Object next() {
            DiagramTreeNode nextDTN = _nextDTN;
            _nextDTN = nextDTN();
            return nextDTN;
```

```
        }

        public void remove() {
            _it.remove();
        }

        private DiagramTreeNode nextDTN() {
            DiagramTreeNode answer = null;

            while (_it.hasNext()) {
                DiagramTreeNode dtn = (DiagramTreeNode)_it.next();
                Diagram nextDiag = dtn.getDiagram();

                if (!DiagramDegeneracyTester.isDegenerate(nextDiag) && dtn.getLeaf()) {
                    answer = dtn;
                    break;
                }
            }

            return answer;
        }
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;

/**
 *
 * @author grouptheory
 */
public class BFS {

    private Node _src;
    private LinkedList _frontier;
    private HashSet _markedNodes;
    private HashSet _markedEdges;
    private HashMap _node2prevEdge;
    private HashMap _edge2prevNode;
    private Diagram _d;

    BFS(Diagram d, Node src) {
        _d = d;
        _src = src;
        _frontier = new LinkedList();
        _markedNodes = new HashSet();
        _markedEdges = new HashSet();
        _node2prevEdge = new HashMap();
        _edge2prevNode = new HashMap();

        _frontier.addLast(src);
        _markedNodes.add(src);
        grow();
    }

    void grow() {
        while (step());
        validate();
    }
```

```
    boolean step() {
        Node nd = (Node)_frontier.removeFirst();
        for (Iterator it = nd.edgeIterator(); it.hasNext(); ) {
            Edge e = (Edge)it.next();
            if (e.getOccupancy() == 2) {
                continue;
            }
            Node peer = e.getOpposite(nd);

            if (_markedNodes.contains(peer)) {
                if ( ! _edge2prevNode.containsKey(e)) {
                    _edge2prevNode.put(e, nd);
                }
                _markedEdges.add(e);
                continue;
            }

            if (_frontier.contains(peer)) {
                if ( ! _edge2prevNode.containsKey(e)) {
                    _edge2prevNode.put(e, nd);
                }
                _markedEdges.add(e);
                continue;
            }

            _frontier.add(peer);
            _markedEdges.add(e);

            _node2prevEdge.put(peer, e);
            _edge2prevNode.put(e, nd);
        }
        _markedNodes.add(nd);
        return _frontier.size()>0;
    }

    boolean reachable(Node nd) {
        _d.validateNode(nd);
        return _markedNodes.contains(nd);
    }

    boolean reachable(Edge e) {
        _d.validateEdge(e);
        return _markedEdges.contains(e);
    }

    Iterator reachableNodesIterator() {
        return _markedNodes.iterator();
```

```
    }

    Iterator reachableEdgesIterator() {
        return _markedEdges.iterator();
    }

    Path getPathFrom(Node nd) {
        _d.validateNode(nd);

        Path p = new Path(nd, _d);

        // System.out.println("Trying to go from "+nd+" to "+_src);
        while (nd != _src) {
            // System.out.println("At node "+nd);

            Edge e = (Edge)_node2prevEdge.get(nd);
            if (e==null) {
                System.out.println("DEBUG BFS\n"+this);
                System.out.println("DEBUG Diagram "+_d+"\n");
                System.out.print("DEBUG Reachable nodes: ");
                for (Iterator it = _markedNodes.iterator(); it.hasNext(); ) {
                    Node rn = (Node)it.next();
                    System.out.print(""+rn.toStringShort()+", ");
                }
                System.out.println("");

                System.out.println("DEBUG Missing entry for node "+nd+"\n");
                throw new RuntimeException("DiagramTreeNode.getPathFrom: traceback from node failed");
            }
            nd = (Node)_edge2prevNode.get(e);
            if (nd==null) {
                System.out.println("DEBUG BFS\n"+this);
                System.out.println("DEBUG Diagram "+_d+"\n");
                System.out.println("DEBUG Missing entry for edge "+e+"\n");
                throw new RuntimeException("DiagramTreeNode.getPathFrom: traceback from edge failed");
            }

            p.append(e);
            p.append(nd);
        }
        return p;
    }


    Path getPathFrom(Edge e) {
        _d.validateEdge(e);
```

```
        Path p = new Path(e, _d);
        Node nd = (Node)_edge2prevNode.get(e);
        p.append(nd);
        while (nd != _src) {
            Edge e2 = (Edge)_node2prevEdge.get(nd);
            nd = (Node)_edge2prevNode.get(e2);
            p.append(e2);
            p.append(nd);
        }
        return p;
    }

    public String toString() {
        String s = "";
        for (Iterator it = _edge2prevNode.entrySet().iterator(); it.hasNext(); ) {
            Map.Entry ent = (Map.Entry)it.next();
            s += "edge "+ent.getKey()+" ==> node "+ent.getValue()+"\n";
        }
        for (Iterator it = _node2prevEdge.entrySet().iterator(); it.hasNext(); ) {
            Map.Entry ent = (Map.Entry)it.next();
            s += "node "+ent.getKey()+" ==> edge "+ent.getValue()+"\n";
        }
        return s;
    }

    public void validate() {
        for (Iterator it = reachableNodesIterator(); it.hasNext();) {
            Node nd = (Node)it.next();
            if (nd == _src) continue;

            Edge e = (Edge)_node2prevEdge.get(nd);
            if (e==null) {
                System.out.println("BAD BFS\n"+this);
                throw new RuntimeException("DiagramTreeNode.getPathFrom: traceback from "+nd+" failed");
            }
        }

        for (Iterator it = reachableEdgesIterator(); it.hasNext();) {
            Edge e = (Edge)it.next();
            Node nd = (Node)_edge2prevNode.get(e);
            if (nd==null) {
                System.out.println("BAD BFS\n"+this);
                throw new RuntimeException("DiagramTreeNode.getPathFrom: traceback from "+e+" failed");
            }
        }
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import equation.QuadraticSystem;
import equation.QuadraticSystemFactory;
import equation.GroupEquation;
import java.util.LinkedList;
import java.util.HashMap;
import utility.CompositeIterator;

/**
 *
 * @author grouptheory
 */
public class CancellationDiagramFactory {

    private static CancellationDiagramFactory _instance;

    private CancellationDiagramFactory() {
    }

    public static CancellationDiagramFactory instance() {
        if (_instance == null) {
            _instance = new CancellationDiagramFactory();
        }
        return _instance;
    }

    public ICancellationDiagramAnalysis newCancellationDiagramAnalysis(GroupEquation problem) {
        return newDiagramProbe(problem);
        // return newDiagramTree(problem);
    }

    public ICancellationDiagramAnalysis newDiagramTree(GroupEquation problem) {

        QuadraticSystem qs;
        qs = QuadraticSystemFactory.instance().newQuadraticSystem(problem);
        GroupEquation problemQuadratic = qs.getEquation();
        HashMap equiv = qs.getEquivalences();

        ExtensionVisitor queryvis = new ExtensionVisitor(problemQuadratic);
        Diagram actual = new Diagram();
        DiagramTreeNode root = new DiagramTreeNode(actual);
```

```
        root.visitedBy(queryvis);

        DiagramTreeNode.CollectionVisitor resultsvis = new DiagramTreeNode.CollectionVisitor();
        root.visitedBy(resultsvis);

        LinkedList diagrams = resultsvis.getDiagramTreeNodeList();

        CancellationDiagramTree analysis = new CancellationDiagramTree(problem, qs, diagrams);
        return analysis;
    }


    public ICancellationDiagramAnalysis newDiagramProbe(GroupEquation problem) {
        QuadraticSystem qs = QuadraticSystemFactory.instance().newQuadraticSystem(problem);
        CancellationDiagramProbe analysis = new CancellationDiagramProbe(problem, qs);
        return analysis;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import java.util.TreeSet;
import java.util.LinkedList;
import java.util.HashMap;
import java.util.Iterator;
import letter.Letter;
import equation.GroupEquation;
import equation.QuadraticSystem;
import utility.CompositeIterator;

/**
 *
 * @author grouptheory
 */
public class CancellationDiagramProbe
        extends AbstractCancellationDiagramAnalysis {

    CancellationDiagramProbe(GroupEquation problem, QuadraticSystem qs) {
        super(problem, qs);
    }

    public Iterator iteratorDiagramTreeNodes() {
        Diagram d = new Diagram();
        GroupEquation problemQuadratic = getQuadraticEquation();
        ComposableDiagramIterator cdi = new ComposableDiagramIterator(null, d, problemQuadratic, 0);
        CompositeIterator compiter = new CompositeIterator(cdi, true);

        // return compiter;
        return new AbstractCancellationDiagramAnalysis.FilteredIterator(compiter);
    }

    public String toString() {
        String s = "";
        s += "CancellationDiagramProbe: ";
        s += super.toString();
        return s;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import java.util.TreeSet;
import java.util.LinkedList;
import java.util.HashMap;
import java.util.Iterator;
import letter.Letter;
import equation.GroupEquation;
import equation.QuadraticSystem;

/**
 *
 * @author grouptheory
 */
public class CancellationDiagramTree
        extends AbstractCancellationDiagramAnalysis {

    private LinkedList _diagrams;

    CancellationDiagramTree(GroupEquation problem, QuadraticSystem qs, LinkedList diagrams) {
        super(problem, qs);

        _diagrams = new LinkedList();
        _diagrams.addAll(diagrams);
    }

    int getDiagramTreeNodesCount() {
        return _diagrams.size();
    }

    public Iterator iteratorDiagramTreeNodes() {
        // return _diagrams.iterator();
        return new AbstractCancellationDiagramAnalysis.FilteredIterator(_diagrams.iterator());
    }

    public String toString() {
        String s = "";
        s += "CancellationDiagramTree: ";
        s += super.toString();

        s+="\n"+_diagrams.size()+" cancellation diagrams enumerated:\n";
        for (Iterator it = _diagrams.iterator();it.hasNext(); ) {
```

```
            DiagramTreeNode dtn2 = (DiagramTreeNode)it.next();
            s+=dtn2.getDiagram().toString();
        }
        s += "\n";
        return s;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import utility.AbstractComposableIterator;
import utility.ComposableIterator;
import equation.GroupEquation;
import letter.Letter;

/**
 *
 * @author grouptheory
 */
public class ComposableDiagramIterator
        extends AbstractComposableIterator
        implements ComposableIterator {

    private Diagram _d;
    private Letter _let;
    private GroupEquation _subeq;
    private ExtensionIterator _eiReading;
    private ExtensionIterator _eiChildren;
    private int _IAmChildNumber;
    private int _YouAreChildNumber;

    public ComposableDiagramIterator(ComposableIterator parent, Diagram d, GroupEquation eq, int childNumber) {
        setParent(parent);

        _d = d;

        if (eq.length() > 0) {
            GroupEquation eq2 = new GroupEquation(eq);
            _let = eq2.popLetter();
            _subeq = eq2;

            if (_subeq.length() > 0) {
                _eiReading = ExtensionIteratorFactory.instance().newExtensionIterator(_d, _let);
                _eiChildren = ExtensionIteratorFactory.instance().newExtensionIterator(_d, _let);
            }
            else {
                _eiReading = ExtensionIteratorFactory.instance().newExtensionIteratorLast(_d, _let);
                _eiChildren = ExtensionIteratorFactory.instance().newExtensionIteratorLast(_d, _let);
            }
        }
```

```
        else {
            throw new RuntimeException("ComposableDiagramIterator.ctor: empty eqn");
        }
        _IAmChildNumber = childNumber;
        _YouAreChildNumber = 0;

        setState(null);
    }

    public ComposableIterator newComposableIterator(ComposableIterator parent) {
        if (_subeq.length() > 0) {
            Extension ex = _eiChildren.next();
            Diagram dnext = ex.apply(_d);
            _YouAreChildNumber++;
            // System.out.println("DEBUG NOT NULL\n");
            return new ComposableDiagramIterator(parent, dnext, _subeq, _YouAreChildNumber);
        }
        else {
            // System.out.println("DEBUG I AM RET NULL\n");
            return null;
        }
    }

    public boolean hasNext() {
        return _eiReading.hasNext();
    }

    public Object next() {
        DiagramTreeNode answer = null;

        Extension ex = _eiReading.next();
        Diagram  dnext = ex.apply(_d);

        ComposableIterator parent = this.getParent();
        if (parent==null) {
            // System.out.println("PARENT NULL");
            answer = new DiagramTreeNode(dnext);
        }
        else {
            // System.out.println("*** not null");
            answer = new DiagramTreeNode((DiagramTreeNode)parent.getState(), _IAmChildNumber, dnext);
        }

        if (_subeq.length() == 0) {
            answer.setLeaf();
        }
```

```
        return answer;
    }

    public void remove() {
        throw new RuntimeException("ComposableDiagramIterator.remove: not implemented");
    }

    public String toString() {
        String s="";
        s += "ComposableDiagramIterator BEGIN\n";
        if (_let==null) {
            s += "    Letter: null\n";
        }
        else {
            s += "    Letter: "+_let.toString() +"\n";
        }
        s += "   Diagram: "+_d.toString()+"";
        s += "ComposableDiagramIterator END\n";
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import java.util.HashMap;
import java.util.Iterator;
import java.util.HashSet;
import java.util.LinkedList;
import letter.Letter;
import letter.Variable;

/**
 *
 * @author grouptheory
 */
public class Diagram {

    private int _nextID;
    private int _nodeCount;
    private HashMap _nodes;

    private int _edgeCount;
    private HashSet _edges;

    private Node _begin;
    private Node _end;

    LinkedList _labeledPaths;

    public Diagram() {
        _nodes = new HashMap();
        _nodeCount = 0;
        _edges = new HashSet();
        _edgeCount = 0;

        _labeledPaths = new LinkedList();

        _nextID = 0;

        _begin = addNode();
        _end = _begin;
    }

    Diagram(Diagram d) {
```

```
        _nodes = new HashMap();
        _nodeCount = 0;
        _edges = new HashSet();
        _edgeCount = 0;
        _labeledPaths = new LinkedList();

        _nextID = d._nextID;

        for (Iterator it = d._nodes.keySet().iterator(); it.hasNext();) {
            Integer id = (Integer)it.next();
            this.addNode(id.intValue());
        }

        for (Iterator it = d._edges.iterator(); it.hasNext();) {
            Edge e = (Edge)it.next();
            Node na = (Node)_nodes.get(new Integer(e.getA().getID()));
            Node nb = (Node)_nodes.get(new Integer(e.getB().getID()));
            this.addEdge(na, nb);
        }

        _begin = (Node)_nodes.get(new Integer(d._begin.getID()));
        _end = (Node)_nodes.get(new Integer(d._end.getID()));

        for (Iterator it = d._labeledPaths.iterator(); it.hasNext();) {
            LabeledPath lp = (LabeledPath)it.next();
            this.addLabeledPath(LabeledPath.project(lp, this));
        }
    }

    public Node getBegin() {
        return _begin;
    }

    public Node getEnd() {
        return _end;
    }

    public boolean isClosed() {
        return this.getBegin()==this.getEnd();
    }

    Node addNode() {
        Node nd = addNode(_nextID);
        _nextID++;
        return nd;
    }
```

```
    private Node addNode(int id) {
        Node nd = new Node(id);
        Node exists = (Node)_nodes.get(id);
        if (exists!=null) {
            throw new RuntimeException("Diagram.addNode: duplicate node "+id);
        }
        _nodes.put(id, nd);
        _nodeCount++;
        return nd;
    }

    Node lookupNode(int id) {
        Node nd = (Node)_nodes.get(id);
        if (nd==null) {
            throw new RuntimeException("Diagram.lookupNode: nonexistent node");
        }
        return nd;
    }

    int getNumberOfNodes() {
        return _nodes.size();
    }

    public Iterator iteratorEdges() {
        return _edges.iterator();
    }

    Iterator iteratorNodes() {
        return _nodes.values().iterator();
    }

    public Iterator iteratorLabeledPaths() {
        return _labeledPaths.iterator();
    }

    public LabeledPath getDual(LabeledPath lp) {
        validateLabeledPath(lp);

        Letter let = lp.getLabel();
        if (let.isConstant()) return null;

        LabeledPath answer = null;
        int found = 0;
        for (Iterator it = _labeledPaths.iterator(); it.hasNext();) {
            LabeledPath lp2 = (LabeledPath)it.next();
            Letter let2 = lp2.getLabel();
```

```
            if ((let==let2) || (let==let2.getInverse())) {
                if (lp2 == lp) continue;
                else {
                    found++;
                    answer = lp2;
                }
            }
        }

        if (found > 1) {
            throw new RuntimeException("Diagram.getDual: variable occurs more than 2 times");
        }
        return answer;
    }

    public LabeledPath[] getPaths(Edge e) {
        validateEdge(e);

        LabeledPath[] lpArray = new LabeledPath[2];

        int found = 0;
        for (Iterator it = _labeledPaths.iterator(); it.hasNext();) {
            LabeledPath lp2 = (LabeledPath)it.next();
            Path p2 = lp2.getPath();
            Letter let2 = lp2.getLabel();

            if (p2.hasEdge(e)) {
                if (found >= 2) {
                    throw new RuntimeException("Diagram.getPaths: more than 2 paths on an edge!");
                }
                lpArray[found]=lp2;
                found++;
            }
        }

        if (found != 2) {
            throw new RuntimeException("Diagram.getPaths: didn’t find 2 paths on an edge!");
        }

        return lpArray;
    }

    public LabeledPath getVariablePath(Variable v) {

        // System.out.println("Searching for "+v+"\n");

        LabeledPath answer = null;
```

```
        int found = 0;
        for (Iterator it = _labeledPaths.iterator(); it.hasNext();) {
            LabeledPath lp2 = (LabeledPath)it.next();
            Letter let2 = lp2.getLabel();

            if ((v==let2) || (v==let2.getInverse())) {
                found++;
                if (answer==null) {
                    answer = lp2;

                    // System.out.println("Found "+lp2+"\n");
                }
            }
        }

        if (found != 2) {
            throw new RuntimeException("Diagram.getDual: equivalent variable occurs != 2 times");
        }
        return answer;
    }

    int getNumberOfEdges() {
        return _edges.size();
    }

    Edge lookupEdge(int id1, int id2) {
        Node nd1 = lookupNode(id1);
        Node nd2 = lookupNode(id2);
        Edge e = nd1.getEdge(nd2);
        return e;
    }

    Edge addEdge(Node a, Node b) {
        validateNode(a);
        validateNode(b);

        int a1=a.degree();
        int b1=b.degree();

        Edge e = a.addEdge(b);
        _edges.add(e);
        _edgeCount++;

        int a2=a.degree();
        int b2=b.degree();
        if ( a2 != a1+1) {
            throw new RuntimeException("Diagram.addEdge: failed on "+a);
```

```
        }
        if ( b2 != b1+1) {
            throw new RuntimeException("Diagram.addEdge: failed on "+b);
        }

        return e;
    }

    void delEdge(Edge e) {
        validateEdge(e);

        Node a = e.getA();
        Node b = e.getB();
        a.delEdge(b);
        b.delEdge(a);

        _edges.remove(e);
        _edgeCount--;
    }

    boolean isCuttableEdge(Edge e) {
        for (Iterator it = _labeledPaths.iterator(); it.hasNext();) {
            LabeledPath lp = (LabeledPath)it.next();
            if (lp.hasEdge(e)) {
                Letter let = lp.getLabel();
                if (let.isConstant()) return false;
            }
        }
        return true;
    }

    Node cutEdge(Edge e) {
        validateEdge(e);

        Node a = e.getA();
        Node b = e.getB();

        Node c = addNode();
        Edge ac = addEdge(a,c);
        Edge bc = addEdge(b,c);
        ac.setOccupancy(e.getOccupancy());
        bc.setOccupancy(e.getOccupancy());

        for (Iterator it = _labeledPaths.iterator(); it.hasNext();) {
            LabeledPath lp = (LabeledPath)it.next();
            lp.cutEdge(e, c);
        }
```

```
        delEdge(e);

        return c;
    }

    void addLabeledPath(LabeledPath lp) {
        lp.validate(this);
        _labeledPaths.addLast(lp);

        for (Iterator it = lp.iteratorEdges(); it.hasNext();) {
            Edge e = (Edge)it.next();
            e.incrementOccupancy();
        }

        _end = lp.getNewEnd();
    }

    public String toString() {
        String s = "";
        s+=">>>>>>>>>>>>>>>>>>>>>>>>>>>>>\n";
        s += "nodes: ";
        for (Iterator it = _nodes.values().iterator(); it.hasNext();) {
            Node nd = (Node)it.next();
            s += nd.toStringShort()+", ";
        }
        s += "\nstart: "+_begin.toStringShort()+" ---> ";
        s += "end: "+_end.toStringShort()+"\n";

        s += "edges: ";

        for (Iterator it = _edges.iterator(); it.hasNext();) {
            Edge e = (Edge)it.next();
            s += e.toString()+", ";
        }
        s += "\n";
        s += "labeled paths:\n";

        for (Iterator it = _labeledPaths.iterator(); it.hasNext();) {
            LabeledPath lp = (LabeledPath)it.next();
            s += "   "+lp.toString()+"\n";
        }
        s+=" <<<\n";

        return s;
    }
```

```
    void validateNode(Node node) {
        Node exists = (Node)_nodes.get(node.getID());
        if (exists==null) {
            throw new RuntimeException("Diagram.validateNode: nonexistent node");
        }
        if (exists!=node) {
            throw new RuntimeException("Diagram.validateNode: bad node");
        }
    }

    void validateEdge(Edge e) {
        if ( ! _edges.contains(e)) {
            throw new RuntimeException("Diagram.validateNode: nonexistent edge");
        }
    }

    void validateLabeledPath(LabeledPath lp) {
        if ( ! _labeledPaths.contains(lp)) {
            throw new RuntimeException("Diagram.validateLabeledPath: nonexistent path");
        }
    }

    void validate() {
        if (_edges.size() != _edgeCount) {
            throw new RuntimeException("Diagram.validate: bad _edgeCount");
        }
        if (_nodes.size() != _nodeCount) {
            throw new RuntimeException("Diagram.validate: bad _nodeCount");
        }
        if ( ! _nodes.containsValue(_begin)) {
            throw new RuntimeException("Diagram.validate: bad _begin");
        }
        if ( ! _nodes.containsValue(_end)) {
            throw new RuntimeException("Diagram.validate: bad _end");
        }
    }


    public interface Decorator {
    }

    private Decorator _decorator;

    Decorator getDecorator() {
        return _decorator;
    }
```

```
    void setDecorator(Decorator dec) {
        _decorator = dec;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import jigglecore.*;
import java.util.Iterator;
import java.util.HashMap;

/**
 *
 * @author grouptheory
 */
public class DiagramDecoratorLayout extends Graph implements Diagram.Decorator {

    public DiagramDecoratorLayout (Diagram cd) {super (); initialize (cd);}
    public DiagramDecoratorLayout (Diagram cd, int d) {super (d); initialize (cd);}

    private void initialize (Diagram cd) {
        int n = cd.getNumberOfNodes();

        // System.out.println("DEBUG DiagramDecoratorLayout n="+n);

        Vertex V [] = new Vertex [n];
        for (int i=0;i<n;i++) {
            V[i] = this.insertVertex();
        }

        // System.out.println("DEBUG DiagramDecoratorLayout e="+cd.getNumberOfEdges());

        for (Iterator it = cd.iteratorEdges(); it.hasNext();) {
            cancellation.Edge e =
                    (cancellation.Edge)it.next();
            insertEdge (V [e.getA().getID()], V [e.getB().getID()]);
        }

        // System.out.println("BEFORE: "+this.toString());
        _vertices = layout(cd);
        // System.out.println("AFTER: "+this.toString());
        // System.out.println("AFTER: "+this.validate(cd));
    }

    private Vertex getVertex(Node nd) {
        Vertex v = (Vertex)_vertices.get(nd.getID());
        if (v==null) {
```

```
            throw new RuntimeException("DiagramLayout.getVertex: unknown Node "+nd);
        }
        return v;

    }

    public double getX(Node nd) {
        Vertex v = getVertex(nd);
        return v.getCoords()[0];
    }

    public double getY(Node nd) {
        Vertex v = getVertex(nd);
        return v.getCoords()[1];
    }

    private HashMap _vertices;

    private final static double LAYOUT_SIDE = 6.0;
    private final static double MAX_ERROR = 0.1;

    private HashMap layout(Diagram cd) {

      _vertices = new HashMap();

      int crossings = 0;

      do {
          _vertices.clear();

          int d = this.getDimensions ();
          int n = this.numberOfVertices;

          double kspring = 10.0;
          double kvv = 1.0;
          double kve = 1.0;

          QuadraticSpringLaw springLaw =
                  new QuadraticSpringLaw (this, kspring);
          InverseSquareVertexVertexRepulsionLaw vvRepulsionLaw =
                  new InverseSquareVertexVertexRepulsionLaw (this, kvv);
          InverseSquareVertexEdgeRepulsionLaw veRepulsionLaw =
                  new InverseSquareVertexEdgeRepulsionLaw (this, kve);

          ForceModel fm = new ForceModel(this);
          fm.addForceLaw (springLaw);
          fm.addForceLaw (vvRepulsionLaw);
```

```
        fm.addForceLaw (veRepulsionLaw);

        // make optimization procedure

        FirstOrderOptimizationProcedure opt = null;
        double acc = 0.5;
        double rt = 0.2;
        opt = new ConjugateGradients (this, fm, acc, rt);
        opt.setConstrained (true);

        // set vertex sizes

        for (int i = 0; i < n; i++) {
            double size [] = this.vertices [i].getSize ();
            size [0] = 10;
            size [1] = 10;
        }

        // scramble vertices

        double w = LAYOUT_SIDE,  h = LAYOUT_SIDE;
        for (int i = 0; i < n; i++) {
            double coords [] = this.vertices [i].getCoords ();
            for (int j = 0; j < d; j++) coords [j] = Math.random () * w;
        }
        double sumX = 0, sumY = 0;
        for (int i = 0; i < n; i++) {
            double coords [] = this.vertices [i].getCoords ();
            sumX += coords [0]; sumY += coords [1];
        }
        for (int i = 0; i < n; i++) {
            Vertex v = this.vertices [i];
            double coords [] = this.vertices [i].getCoords ();
            coords [0] += (w / 2) - (sumX / n);
            coords [1] += (h / 2) - (sumY / n);
        }

        // Optimization here

        int iter = 0;
        double val = 999.0;
        do {
            val = opt.improveGraph();
            // System.out.println("iter:"+iter+", err = "+val);
            iter++;
        }
        while(val > MAX_ERROR);
```

```
            // renormalize vertex coordinates

            double minx=999, miny=999, maxx=-999, maxy=-999;

            for (int i = 0; i < n; i++) {
                double coords [] = this.vertices [i].getCoords ();
                if (coords [0]>maxx) maxx=coords[0];
                if (coords [1]>maxy) maxy=coords[1];
                if (coords [0]<minx) minx=coords[0];
                if (coords [1]<miny) miny=coords[1];
            }

            for (int i = 0; i < n; i++) {
                double coords [] = this.vertices [i].getCoords ();
                coords[0] = (coords[0]-minx) * LAYOUT_SIDE/(maxx-minx);
                coords[1] = (coords[1]-miny) * LAYOUT_SIDE/(maxy-miny);
                _vertices.put(new Integer(i), this.vertices [i]);
            }

            crossings = Intersector.crossingNumber(this);

            /*
            if (crossings == 0) {
                System.out.println("Diagram "+cd.toString());
                System.out.println(this.validate(cd));
                System.out.println("Converged after "+iter+" iterations, err = "+val+", crossings = "+crossings);
            }
            */
        }
        while (crossings > 0);

        return _vertices;
    }

    public String toString() {
        String s = "";
        int n = this.numberOfVertices;
        for (int i = 0; i < n; i++) {
           double coords [] = this.vertices [i].getCoords ();
           s += "Vertex "+i+" @ "+coords[0]+" , "+coords[1]+"\n";
        }
        return s;
    }


    public String validate(Diagram d) {
```

```
        String s = "";
        for (Iterator it = d.iteratorNodes(); it.hasNext();) {
            Node nd = (Node)it.next();
            s += "Node "+nd.getID()+" @ "+getX(nd)+" , "+this.getY(nd)+"\n";
        }
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

/**
 *
 * @author grouptheory
 */
public class DiagramDegeneracyTester {

    public static boolean isDegenerate(Diagram d) {

        if ( ! d.isClosed()) return true;
        else
            return false;
    }

}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import letter.Letter;
import equation.GroupEquation;
import java.util.LinkedList;
import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
public class DiagramTreeNode {
    private Diagram _d;
    private ExtensionIterator _ei;
    private LinkedList _children;
    private String _name;
    private int _depth;
    private boolean _leaf;

    DiagramTreeNode(Diagram d) {
        _d = d;
        _children = new LinkedList();
        _name = "root";
        _depth = 0;
        _leaf = false;
    }

    DiagramTreeNode(DiagramTreeNode parent, int childNumber, Diagram d) {
        _d = d;
        _children = new LinkedList();
        _name = parent._name+"."+childNumber;
        _depth = parent._depth+1;
        _leaf = false;
    }

    public Diagram getDiagram() {
        return _d;
    }

    public String getName() {
        return _name;
    }
```

```java
    void setLeaf() {
        _leaf = true;
    }

    public boolean getLeaf() {
        return _leaf;
    }

    int getDepth() {
        return _depth;
    }

    void extend(Letter let, boolean last) {
        if (_ei != null) {
            throw new RuntimeException("DiagramTreeNode.extend: multiple calls to extend");
        }

        if (last) {
            _ei = ExtensionIteratorFactory.instance().newExtensionIteratorLast(_d, let);
        }
        else {
            _ei = ExtensionIteratorFactory.instance().newExtensionIterator(_d, let);
        }
        int childNumber = 0;

        // System.out.println("DEBUG OUTSIDE LOOP Extension of _d="+_d);

        for ( ;_ei.hasNext(); ) {
            Extension ex = _ei.next();

            // System.out.println("DEBUG INSIDE LOOP Extension of _d="+_d);
            // System.out.println("DEBUG Extension: "+ex);

            Diagram d2 = ex.apply(_d);
            DiagramTreeNode dtn2 = new DiagramTreeNode(this, childNumber, d2);
            if (last) {
                dtn2.setLeaf();
            }
            _children.add(dtn2);
            childNumber++;
        }
    }

    public String toString() {
        String s = "DiagramTreeNode BEGIN\n";
        s += _d.toString();
```

```
        s += "DiagramTreeNode END\n";
        return s;
    }

    void visitedBy(ExtensionVisitor ev) {
        ev.visit(this);

        // DEBUG
        // this.visitedBy(new DiagramTreeNode.ConsoleVisitor());

        Iterator it = _children.iterator();
        for ( ;it.hasNext(); ) {
            DiagramTreeNode dtn2 = (DiagramTreeNode)it.next();
            ExtensionVisitor ev2 = ev.makeSubVisitor();
            dtn2.visitedBy(ev2);
        }
    }

    void visitedBy(ConsoleVisitor pv) {
        pv.visit(this);
        Iterator it = _children.iterator();
        for ( ;it.hasNext(); ) {
            DiagramTreeNode dtn2 = (DiagramTreeNode)it.next();
            dtn2.visitedBy(pv);
        }
    }

    void visitedBy(CollectionVisitor cv) {
        cv.visit(this);
        Iterator it = _children.iterator();
        for ( ;it.hasNext(); ) {
            DiagramTreeNode dtn2 = (DiagramTreeNode)it.next();
            dtn2.visitedBy(cv);
        }
    }

    static class ConsoleVisitor {
        ConsoleVisitor() { }
        void visit(DiagramTreeNode dtn) {
            System.out.println("DiagramTreeNode: "+dtn._name+"(depth="+dtn._depth+")");
            System.out.println(dtn._d);
        }
    }

    static class CollectionVisitor {
        private LinkedList _treenodes;
```

```
        CollectionVisitor() {
            _treenodes = new LinkedList();
        }

        void visit(DiagramTreeNode dtn) {
            Node begin = dtn._d.getBegin();
            Node end = dtn._d.getEnd();
            if ((begin==end) && dtn.getLeaf()) {
                _treenodes.add(dtn);
            }
        }

        public String toString() {
            String s = "";
            Iterator it = _treenodes.iterator();
            for ( ;it.hasNext(); ) {
                DiagramTreeNode dtn2 = (DiagramTreeNode)it.next();
                s+=dtn2._d.toString();
            }
            s+=""+_treenodes.size()+" cancellation diagrams were enumerated.\n";
            return s;
        }

        public LinkedList getDiagramTreeNodeList() {
            return _treenodes;
        }
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import java.util.Comparator;

/**
 *
 * @author grouptheory
 */
public class DiagramTreeNodeComparator implements Comparator {

    public int compare(Object o1, Object o2) {
        DiagramTreeNode d1 = (DiagramTreeNode)o1;
        DiagramTreeNode d2 = (DiagramTreeNode)o2;


        if (d1.hashCode() < d2.hashCode()) {
            return -1;
        }
        else if (d1.hashCode() > d2.hashCode()) {
            return +1;
        }
        else {
            return 0;
        }
    }

    public boolean equals(Object obj) {
        return obj==this;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

/**
 *
 * @author grouptheory
 */
public class Edge {
    private Node _a, _b;
    private int _occupancy;
    private Edge _reverse;

    Edge(Node a, Node b) {
        if (a==b) {
            throw new RuntimeException("Edge.ctor: loop edge");
        }
        _a = a;
        _b = b;
        _occupancy = 0;
        _reverse = null;
    }

    void setReverse(Edge rev) {
        _reverse = rev;
    }

    Node getA() {
        return _a;
    }

    Node getB() {
        return _b;
    }

    int getOccupancy() {
        return _occupancy;
    }

    void setOccupancy(int occupancy) {
        _occupancy = occupancy;
    }

    void incrementOccupancy() {
```

```
        _occupancy++;
    }

    Node getOpposite(Node x) {
        if (_a == x) return _b;
        else if(_b == x) return _a;
        else {
            throw new RuntimeException("Edge.getOpposite: bad edge");
        }
    }

    public String toString() {
        String s = "";
        s += "(" + _a.toStringShort() + "," + _b.toStringShort() + ")";
        if (getOccupancy()==2) s+="*";
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import letter.Letter;

/**
 *
 * @author grouptheory
 */
public abstract class Extension {
    protected Letter _label;

    protected Extension(Letter label) {
        _label = label;
    }

    abstract Diagram apply(Diagram d);
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import java.util.LinkedList;
import java.util.Iterator;
import java.util.ListIterator;

/**
 *
 * @author grouptheory
 */
public class ExtensionIterator {

    private LinkedList _extensions;
    private Iterator _it;

    ExtensionIterator(LinkedList exts) {
        _extensions = new LinkedList();
        _extensions.addAll(exts);
        _it = _extensions.iterator();
    }

    boolean hasNext() {
        return _it.hasNext();
    }

    Extension next() {
        return (Extension)_it.next();
    }

    public String toString() {
        String s = "";
        for (Iterator it = _extensions.iterator(); it.hasNext();) {
            s += ((Extension)it.next()).toString()+"\n";
        }
        return s;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import letter.Letter;
import java.util.LinkedList;
import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
public class ExtensionIteratorFactory {

    private static ExtensionIteratorFactory _instance;

    private ExtensionIteratorFactory() {
    }

    public static ExtensionIteratorFactory instance() {
        if (_instance == null) {
            _instance = new ExtensionIteratorFactory();
        }
        return _instance;
    }

    ExtensionIterator newExtensionIteratorLast(Diagram d, Letter let) {
        LinkedList exts = new LinkedList();

        Node dst = d.getEnd();
        BFS bfs = new BFS(d,dst);
        Node src = d.getBegin();

        Path p1 = bfs.getPathFrom(src);

        if (let.isConstant()) {
            // constants
            if (p1.length()==1) {
                Extension ex1=new ExtensionToVertexExact(let, src);
                exts.add(ex1);
            }
        }
        else {
            // variables
```

```
            if (p1.length() > 0) {
                Extension ex1=new ExtensionToVertexExact(let, src);
                exts.add(ex1);
            }
        }

        return new ExtensionIterator(exts);
    }

    ExtensionIterator newExtensionIterator(Diagram d, Letter let) {
        LinkedList exts = new LinkedList();

        Node dst = d.getEnd();
        BFS bfs = new BFS(d,dst);

        Iterator srcNodeIt = bfs.reachableNodesIterator();
        for ( ;srcNodeIt.hasNext(); ) {
            Node src = (Node)srcNodeIt.next();
            Path p1 = bfs.getPathFrom(src);

            if (let.isConstant()) {
                // constants
                if (p1.length()==1) {
                    Extension ex1=new ExtensionToVertexExact(let, src);
                    exts.add(ex1);
                }

                if (p1.length()==0) {
                    Extension ex1=new ExtensionToVertexSpur(let, src);
                    exts.add(ex1);
                }
            }
            else {
                // variables
                if (p1.length() > 0) {
                    Extension ex1=new ExtensionToVertexExact(let, src);
                    exts.add(ex1);
                }

                Extension ex2=new ExtensionToVertexSpur(let, src);
                exts.add(ex2);
            }
        }


        Iterator srcEdgeIt = bfs.reachableEdgesIterator();
        for ( ;srcEdgeIt.hasNext(); ) {
```

```
            Edge src = (Edge)srcEdgeIt.next();

            if ( ! d.isCuttableEdge(src)) {
                continue;
            }

            Path p1 = bfs.getPathFrom(src);

            if (let.isConstant()) {
                // constants
                if (p1.length()==1) {
                    Extension ex1=new ExtensionToEdgeExact(let, src);
                    exts.add(ex1);
                }
            }
            else {
                // variables
                Extension ex1=new ExtensionToEdgeExact(let, src);
                exts.add(ex1);

                Extension ex2=new ExtensionToEdgeSpur(let, src);
                exts.add(ex2);
            }
        }

        return new ExtensionIterator(exts);
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import letter.Letter;

/**
 *
 * @author grouptheory
 */
public class ExtensionToEdgeExact extends Extension {
    private static String NAME = "ExtensionToEdgeExact";
    private Edge _srcEdge;

    ExtensionToEdgeExact(Letter label, Edge src) {
        super(label);
        _srcEdge = src;
    }

    Diagram apply(Diagram d) {
        Diagram d2 = new Diagram(d);
        Node end = d2.getEnd();
        Edge src = d2.lookupEdge(_srcEdge.getA().getID(), _srcEdge.getB().getID());
        Node cutNode = d2.cutEdge(src);

        BFS bfs = new BFS(d2, end);
        Path p = bfs.getPathFrom(cutNode);
        LabeledPath lp = new LabeledPath(_label, p);
        d2.addLabeledPath(lp);
        return d2;
    }

    public String toString() {
         return ""+NAME+"->"+_srcEdge+"";
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import letter.Letter;

/**
 *
 * @author grouptheory
 */
public class ExtensionToEdgeSpur extends Extension {
    private static String NAME = "ExtensionToEdgeSpur";
    private Edge _srcEdge;

    ExtensionToEdgeSpur(Letter label, Edge src) {
        super(label);
        _srcEdge = src;
    }

    Diagram apply(Diagram d) {
        Diagram d2 = new Diagram(d);
        Node end = d2.getEnd();
        Edge src = d2.lookupEdge(_srcEdge.getA().getID(), _srcEdge.getB().getID());
        Node cutNode = d2.cutEdge(src);
        Node spurNode = d2.addNode();
        d2.addEdge(cutNode, spurNode);

        BFS bfs = new BFS(d2, end);
        Path p = bfs.getPathFrom(spurNode);
        LabeledPath lp = new LabeledPath(_label, p);
        d2.addLabeledPath(lp);
        return d2;
    }

    public String toString() {
         return ""+NAME+"->"+_srcEdge+"";
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import letter.Letter;

/**
 *
 * @author grouptheory
 */
public class ExtensionToVertexExact extends Extension {
    private static String NAME = "ExtensionToVertexExact";
    private Node _srcNode;

    ExtensionToVertexExact(Letter label, Node src) {
        super(label);
        _srcNode = src;
    }

    Diagram apply(Diagram d) {
        Diagram d2 = new Diagram(d);
        Node end = d2.getEnd();
        Node src = d2.lookupNode(_srcNode.getID());

        BFS bfs = new BFS(d2, end);
        Path p = bfs.getPathFrom(src);
        LabeledPath lp = new LabeledPath(_label, p);
        d2.addLabeledPath(lp);
        return d2;
    }

    public String toString() {
         return ""+NAME+"->"+_srcNode.toStringShort()+"";
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import letter.Letter;

/**
 *
 * @author grouptheory
 */
public class ExtensionToVertexSpur extends Extension {
    private static String NAME = "ExtensionToVertexSpur";
    private Node _srcNode;

    ExtensionToVertexSpur(Letter label, Node src) {
        super(label);
        _srcNode = src;
    }

    Diagram apply(Diagram d) {
        Diagram d2 = new Diagram(d);
        Node end = d2.getEnd();
        Node spurNode = d2.addNode();
        Node src = d2.lookupNode(_srcNode.getID());
        d2.addEdge(src, spurNode);

        // System.out.println("DEBUG ExtensionToVertexSpur "+d2);

        BFS bfs = new BFS(d2, end);
        Path p = bfs.getPathFrom(spurNode);
        LabeledPath lp = new LabeledPath(_label, p);
        d2.addLabeledPath(lp);
        return d2;
    }

    public String toString() {
         return ""+NAME+"->"+_srcNode.toStringShort()+"";
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;


import letter.Letter;
import equation.GroupEquation;
import java.util.LinkedList;
import java.util.Iterator;

/**
 *
 * @author grouptheory
 */

class ExtensionVisitor {
    private int _maxdepth;
    private GroupEquation _eq;

    ExtensionVisitor(GroupEquation eq) {
        _eq = eq;
        _maxdepth = eq.length()+1;
    }

    ExtensionVisitor(GroupEquation eq, int maxdepth) {
        _eq = eq;
        _maxdepth = maxdepth;
    }

    ExtensionVisitor makeSubVisitor() {
        GroupEquation eq2 = new GroupEquation(_eq);
        eq2.popLetter();
        ExtensionVisitor ev2 = new ExtensionVisitor(eq2, _maxdepth);
        return ev2;
    }

    void visit(DiagramTreeNode dtn) {
        if (dtn.getDepth() < _maxdepth) {
            if (_eq.length()>0) {
                GroupEquation eq2 = new GroupEquation(_eq);
                Letter let = eq2.popLetter();

                boolean last = false;
                if (_eq.length() == 1) last = true;
```

```
                dtn.extend(let, last);
            }
            else {
                dtn.setLeaf();
            }
        }
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import java.util.TreeSet;
import java.util.LinkedList;
import java.util.HashMap;
import java.util.Iterator;
import letter.Letter;
import equation.GroupEquation;
import equation.QuadraticSystem;
/**
 *
 * @author grouptheory
 */
public interface ICancellationDiagramAnalysis {

    public void addDecorator(ICancellationDiagramAnalysisDecorator dec);
    public Iterator iteratorDecorators();

    public Iterator iteratorDiagramTreeNodes();

    public GroupEquation getProblem();
    public QuadraticSystem getQuadraticSystem();
    public GroupEquation getQuadraticEquation() ;
    public HashMap getEquivalences();
    public String toString();
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

/**
 *
 * @author grouptheory
 */
public interface ICancellationDiagramAnalysisDecorator {

    public String texify(ICancellationDiagramAnalysis analysis, DiagramTreeNode dtn);
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import letter.Letter;
import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
public class LabeledPath {
    private Path _path;
    private Letter _label;

    LabeledPath(Letter label, Path path) {
        _label = label;
        _path = path;
    }

    static LabeledPath project(LabeledPath lp, Diagram d2) {
        LabeledPath lp2 = new LabeledPath(lp._label, lp._path, d2);
        return lp2;
    }

    private LabeledPath(Letter label, Path path, Diagram d2) {
        _label = label;
        _path = Path.project(path, d2);
    }

    public int length() {
        return _path.length();
    }

    void cutEdge(Edge e, Node insert) {
        if (this.hasEdge(e)) {
            _path.cutEdge(e, insert);
        }
    }

    boolean hasEdge(Edge e) {
        return _path.hasEdge(e);
    }
```

```
    public int getEdgeIndex(Edge e) {
        return _path.getEdgeIndex(e);
    }

    Node getNewEnd() {
        return _path.getSrcNode();
    }

    public Path getPath() {
        return _path;
    }

    public Letter getLabel() {
        return _label;
    }

    Iterator iteratorEdges() {
        return _path.iteratorEdges();
    }

    public String toString() {
        String s = "{ "+_label+" :: "+_path+" }";
        return s;
    }

    void validate(Diagram d) {
        _path.validate(d);
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import java.util.Formatter;
import java.util.LinkedList;
import java.util.Iterator;
import java.util.HashMap;
import letter.Letter;

/**
 *
 * @author grouptheory
 */
public class Latex {

    private static Latex _instance;

    private Latex() {
    }

    public static Latex instance() {
        if (_instance == null) {
            _instance = new Latex();
        }
        return _instance;
    }

    private static final double OFFSET_PATHS = 0.25;

    static final int SYMBOL_LEN = 4;
    static final String FORMAT_STR = "%.2f";
    static Formatter _formatter = new Formatter();

    public String renderDiagram(Diagram d) {

        DiagramDecoratorLayout layout;
        layout = (DiagramDecoratorLayout)d.getDecorator();
        if (layout == null) {
            layout = new DiagramDecoratorLayout(d);
            d.setDecorator(layout);
        }

        String str = "";
```

```
        str+="\\begin{center}\n";
        str+="\\begin{pspicture}(-0.5,-0.5)(6.5,6.5)\n";
        str+="{\\psset{fillstyle=ccslope,slopebegin=yellow!40,slopeend=gray}\n";

        for (Iterator it = d.iteratorNodes(); it.hasNext();) {
            Node nd = (Node)it.next();
            double x = layout.getX(nd);
            double y = layout.getY(nd);
            _formatter = new Formatter();
            String xstr = ""+_formatter.format(FORMAT_STR, x);
            _formatter = new Formatter();
            String ystr = ""+_formatter.format(FORMAT_STR, y);

            int id = nd.getID();
            str+="\\cnodeput("+xstr+","+ystr+"){"+id+"}{\\strut\\boldmath$"+id+"$}\n";
        }

        str+="}\n";
        str+="\\newcommand\\arc[3]{%\n";
        str+="  \\ncline{#1}{#2}{#3}\n";
        str+="}\n";

        for (Iterator it=d.iteratorEdges(); it.hasNext();) {
            Edge e = (Edge)it.next();
            str += "\\arc{-}{"+e.getA().getID()+"}{"+e.getB().getID()+"}{}\n";
        }

        HashMap canonical2covered = new HashMap();

        for (Iterator it=d.iteratorLabeledPaths();it.hasNext();) {

            LabeledPath lp = (LabeledPath)it.next();

            Letter let = lp.getLabel();
            Path p = lp.getPath();

            boolean line = (p.length()==1);
            String arrowhead = "{|->>}";
            if (!let.isPositive()) {
                arrowhead = "{<<-|}";
            }

            String colorStr="";
            if (let.isConstant()) {
                if (let.modulus(3)==0) {
                    colorStr="yellow";
                }
```

```
            else if (let.modulus(3)==1) {
                colorStr="blue";
            }
            else if (let.modulus(3)==2) {
                colorStr="green";
            }
        }
        else {
            colorStr="red";
        }

        if (!line) str+="\\pscurve";
        else str+="\\psline";

        str+="[linecolor="+colorStr+"]"+arrowhead;

        LinkedList curves = new LinkedList();

        int midpoint = (p.length()+1)/2;
        int segnum = 1;
        boolean virgin=true;
        String labelStr = "";
        boolean prevcovered=false;

        Node s = p.getSrcNode();

        for (Iterator it2=p.iteratorEdges(); it2.hasNext();){
            Edge e=(Edge)it2.next();

            Node t = e.getOpposite(s);

            double xs = layout.getX(s);
            double ys = layout.getY(s);
            double xt = layout.getX(t);
            double yt = layout.getY(t);
            double angle = Math.atan2(yt-ys, xt-xs);

            boolean covered = (canonical2covered.get(e)!=null);
            angle+=(Math.PI/2.0);
            if ((covered!=prevcovered) && (!virgin)) {
                curves.removeLast();
            }
            prevcovered=covered;

            double dy = OFFSET_PATHS*Math.sin(angle);
            double dx = OFFSET_PATHS*Math.cos(angle);
            xs += dx;
```

```
        ys += dy;
        xt += dx;
        yt += dy;

        _formatter = new Formatter();
        String xsstr = ""+_formatter.format(FORMAT_STR, xs);
        _formatter = new Formatter();
        String ysstr = ""+_formatter.format(FORMAT_STR, ys);
        _formatter = new Formatter();
        String xtstr = ""+_formatter.format(FORMAT_STR, xt);
        _formatter = new Formatter();
        String ytstr = ""+_formatter.format(FORMAT_STR, yt);
        if (virgin) {
            curves.addLast("("+xsstr+","+ysstr+")");
            virgin = false;
        }
        double xm = (xs+xt)/2.0;
        double ym = (ys+yt)/2.0;
        _formatter = new Formatter();
        String xmstr = ""+_formatter.format(FORMAT_STR, xm);
        _formatter = new Formatter();
        String ymstr = ""+_formatter.format(FORMAT_STR, ym);
        if (segnum==midpoint) {
            String letStr = letter.Latex.instance().render(let);
            if (!let.isPositive()) {
                letStr = letter.Latex.instance().render(let.getInverse());
            }

            angle-=(Math.PI/2.0);
            if (angle>Math.PI/2.0) angle-=Math.PI;
            if (angle<-Math.PI/2.0) angle+=Math.PI;
            double deg = (angle*180.0/Math.PI);
            while (deg < 0) deg+=360.0;
            _formatter = new Formatter();
            String degstr = ""+_formatter.format("%d", (int)deg);
            labelStr="\\rput{"+
                    degstr
                    +"}("+xmstr+","+ymstr+"){$"+letStr+"$}";
        }
        curves.addLast("("+xmstr+","+ymstr+")");
        curves.addLast("("+xtstr+","+ytstr+")");
        canonical2covered.put(e, Boolean.TRUE);
        segnum++;

        s = t;
    }
```

```
            for(Iterator it3=curves.iterator();it3.hasNext();) {
                String str2 = (String)it3.next();
                str+=str2;
            }
            str += labelStr+"\n";
        }

        str+="\\end{pspicture}\n";
        str+="\\end{center}\n";


        str+="\\begin{center}\n";
        str+="\\begin{tabular}{|ll|}\n";
        str+="\\hline\n";
        for (Iterator it=d.iteratorLabeledPaths();it.hasNext();) {
            LabeledPath lp = (LabeledPath)it.next();
            Letter let = lp.getLabel();
            Path p = lp.getPath();

            str+="$"+letter.Latex.instance().render(let)+"$";
            str+=" & ";


            Node s = p.getSrcNode();
            Node t = null;
            str += "$";
            for (Iterator it2=p.iteratorEdges(); it2.hasNext();){
                Edge e=(Edge)it2.next();

                t = e.getOpposite(s);

                str += ""+s.getID()+"\\leftarrow ";

                s = t;
            }
            str += ""+t.getID()+"";
            str += "$";

            str+="\\\\\n";
        }
        str+="\\hline\n";
        str+="\\end{tabular}\n";
        str+="\\end{center}\n";
        return str;
    }
```

```
    public String renderCancellationDiagramAnalysis(ICancellationDiagramAnalysis analysis) {
        String s = "";
        s += "We report on the cancellation diagrams of ";
        s += "$";
        s += equation.Latex.instance().renderGroupEquation(analysis.getProblem());
        s += "$";

        s += ",\n";
        s += "which can be re-expressed as a quadratic system ";
        s += "$";
        s += equation.Latex.instance().renderQuadraticSystem(analysis.getQuadraticSystem());
        s += "$";
        s += ".\n";
        s += "Below, we list the possible cancellation diagrams and then represent each as a generalized equation (GE).\n\
n";
        for (Iterator it = analysis.iteratorDiagramTreeNodes(); it.hasNext();) {
            DiagramTreeNode dtn = (DiagramTreeNode)it.next();

            /*
            if (DiagramDegeneracyTester.isDegenerate(dtn.getDiagram())) {
                continue;
            }

            if (!dtn.getLeaf()) {
                // this can happen if |w| is odd and
                // we return back to 1 with at the
                // penultimate symbol.
                continue;
            }
            */

            s += "\\newpage ";
            s += "\n\n{\\bf Cancellation diagram ";
            s+=dtn.getName();
            s += ":}\n";

            // System.out.println(dtn.getName());

            if (params.MKParams.FLAG_REPORT_CANCELLATION_PICTURES) {
                s+=this.renderDiagram(dtn.getDiagram());
            }

            for (Iterator itdec=analysis.iteratorDecorators(); itdec.hasNext();) {
                ICancellationDiagramAnalysisDecorator dec =
                        (ICancellationDiagramAnalysisDecorator)itdec.next();
                s += dec.texify(analysis, dtn);
            }
```

```
        }
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import java.util.Iterator;
import letter.LetterFactory;
import letter.Letter;
import equation.GroupEquation;

/**
 *
 * @author grouptheory
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

        /*
        Diagram d = new Diagram();

        Node n0 = d.getBegin();
        Node n1 = d.addNode();
        Node n2 = d.addNode();
        Node n3 = d.addNode();
        Node n4 = d.addNode();
        Node n5 = d.addNode();
        Edge e1 = d.addEdge(n0, n1);
        Edge e2 = d.addEdge(n1, n2);
        Edge e3 = d.addEdge(n2, n3);
        Edge e4 = d.addEdge(n3, n4);
        Edge e5 = d.addEdge(n4, n5);
        Edge e6 = d.addEdge(n5, n0);

        System.out.println("\nTEST diagram construction:\n");
        System.out.println("d: "+d);

        Node dst = n0;
        BFS b1 = new BFS(d,dst);

        System.out.println("\nTEST node path routing:\n");
```

```
        Iterator srcNodeIt = b1.reachableNodesIterator();
        for ( ;srcNodeIt.hasNext(); ) {
            Node src = (Node)srcNodeIt.next();
            Path p1 = b1.getPathFrom(src);
            System.out.println("from: "+src+" to: "+dst+" === path: "+p1);
        }

        System.out.println("\nTEST edge path routing:\n");
        Iterator srcEdgeIt = b1.reachableEdgesIterator();
        for ( ;srcEdgeIt.hasNext(); ) {
            Edge e = (Edge)srcEdgeIt.next();
            Path p1 = b1.getPathFrom(e);
            System.out.println("from: "+e+" to: "+dst+" === path: "+p1);
        }

        System.out.println("\nTEST extensions by variable:\n");
        Letter varlet = LetterFactory.instance().getVariable(1, Boolean.TRUE);
        ExtensionIterator eit1 = ExtensionIteratorFactory.instance().newExtensionIterator(d, varlet);
        System.out.println(eit1.toString());

        System.out.println("\nTEST extensions by constant:\n");
        Letter constlet = LetterFactory.instance().getConstant(1, Boolean.TRUE);
        ExtensionIterator eit2 = ExtensionIteratorFactory.instance().newExtensionIterator(d, constlet);
        System.out.println(eit2.toString());

        System.out.println("\nTEST node path routing from "+n2+" pre Diagram edge cut\n");
        Path px = b1.getPathFrom(n2);
        System.out.println("pre cut:\n"+px);

        Edge ex = d.lookupEdge(n1.getID(), n2.getID());
        System.out.println("\nTEST cutting Diagram edge "+ex+"\n");
        System.out.println("pre cut:\n"+d);
        Node newNode = d.cutEdge(ex);
        System.out.println("post cut:\n"+d);

        System.out.println("\nTEST cloning cut Diagram:\n");
        System.out.println("d: "+d);
        Diagram dcopy = new Diagram(d);
        System.out.println("dcopy: "+dcopy);

        System.out.println("\nTEST node path routing from "+n2+" post Diagram edge cut "+ex+"\n");
        BFS b2 = new BFS(d,n0);
        px = b2.getPathFrom(n2);
        System.out.println("post cut:\n"+px);

        System.out.println("\nTEST cloning post path routing:\n");
        System.out.println("d: "+d);
```

```
        Diagram d2 = new Diagram(d);
        System.out.println("d2: "+d2);

        System.out.println("\nTEST path projection\n");
        System.out.println("path in d:\n"+px);
        Path px2 = Path.project(px, d2);
        System.out.println("path in d2:\n"+px2);

        System.out.println("\nTEST adding labeled path\n");
        System.out.println("pre add:\n");
        System.out.println("d2: "+d2);
        d2.addLabeledPath(new LabeledPath(varlet, px2));
        System.out.println("\npost addition: "+px+"\n");
        System.out.println("d2: "+d2);

        System.out.println("\nTEST cloning with labelled paths:\n");
        Diagram d3 = new Diagram(d2);
        System.out.println("d3: "+d3);

        Diagram tree = new Diagram();

        Node v0 = tree.getBegin();
        Node v1 = tree.addNode();
        Node v2 = tree.addNode();
        Node v3 = tree.addNode();
        Edge f1 = tree.addEdge(v0, v1);
        Edge f2 = tree.addEdge(v1, v2);
        Edge f3 = tree.addEdge(v2, v3);

        System.out.println("\nTEST DiagramTreeNode:\n");
        DiagramTreeNode dtn = new DiagramTreeNode(tree);
        dtn.visitedBy(new DiagramTreeNode.ConsoleVisitor());

        System.out.println("\nTEST DiagramTreeNode var extension\n");
        dtn.extend(varlet);
        dtn.visitedBy(new DiagramTreeNode.ConsoleVisitor());
        System.out.println("\nThere should be 3+4+3+3=13 (i.e. 0-12)\n");


        System.out.println("\nTEST DiagramTreeNode off same diagram:\n");
        DiagramTreeNode dtn2 = new DiagramTreeNode(tree);
        System.out.println("\nOriginal:\n");
        dtn.visitedBy(new DiagramTreeNode.ConsoleVisitor());
        System.out.println("\nNew:\n");
        dtn2.visitedBy(new DiagramTreeNode.ConsoleVisitor());

        System.out.println("\nTEST DiagramTreeNode const extension\n");
```

```
        dtn2.extend(constlet);
        dtn2.visitedBy(new DiagramTreeNode.ConsoleVisitor());
        System.out.println("\nThere should be 1+1+1 (i.e. 0-2)\n");




        GroupEquation eqn = new GroupEquation("z1+.c1+.z2+.");
        DiagramTreeNode.ExtensionVisitor vis = new DiagramTreeNode.ExtensionVisitor(eqn, 3);

        System.out.println("\nTEST Another DiagramTreeNode off same diagram:\n");
        DiagramTreeNode dtn3 = new DiagramTreeNode(tree);
        System.out.println("\nTree before:\n");
        dtn3.visitedBy(new DiagramTreeNode.ConsoleVisitor());

        dtn3.visitedBy(vis);

        System.out.println("\nTree after:\n");
        dtn3.visitedBy(new DiagramTreeNode.ConsoleVisitor());

        //************************************************
        //************************************************

        System.out.println("\nTEST Real-world EQUATION:\n");

        GroupEquation problem = new GroupEquation("z1+.c1+.z2+.");
        System.out.println("equation: "+problem+" = 1\n");

        DiagramTreeNode.ExtensionVisitor queryvis = new DiagramTreeNode.ExtensionVisitor(problem);
        Diagram actual = new Diagram();
        DiagramTreeNode root = new DiagramTreeNode(actual);

        root.visitedBy(queryvis);

        DiagramTreeNode.CollectionVisitor resultsvis = new DiagramTreeNode.CollectionVisitor();
        root.visitedBy(resultsvis);

        System.out.println("\nTEST Real-world test RESULTS\n");
        System.out.println(""+resultsvis.toString());

*/

        //************************************************
        //************************************************

        System.out.println("\nTEST Real-world EQUATION TEST:\n");
```

```
        GroupEquation prob = new GroupEquation("z1+.c1+.z1+.c2+.z1+.");
        System.out.println("Original Equation: "+prob+" = 1\n");

        ICancellationDiagramAnalysis analysis =
                CancellationDiagramFactory.instance().newDiagramTree(prob);

        System.out.println("Analysis: \n\n");
        System.out.println(analysis.toString());

        System.out.println(Latex.instance().renderCancellationDiagramAnalysis(analysis));
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import java.util.Iterator;
import letter.LetterFactory;
import letter.Letter;
import equation.GroupEquation;
import equation.QuadraticSystem;
import equation.QuadraticSystemFactory;
import utility.CompositeIterator;

/**
 *
 * @author grouptheory
 */
public class Main2 {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        System.out.println("\nTEST COMPOSABLE ITERATORS:\n");

        GroupEquation problem = new GroupEquation("z1+.c1+.z2+.c2+.z1-.c3+.z2-.");
        //GroupEquation problem = new GroupEquation("z1+.z3+.z1+.z1+.z3+.z1+.");

        System.out.println("Original Equation: "+problem+" = 1\n");

        QuadraticSystem qs = QuadraticSystemFactory.instance().newQuadraticSystem(problem);

        GroupEquation problemQuadratic = qs.getEquation();
        Diagram d = new Diagram();

        ComposableDiagramIterator cdi = new ComposableDiagramIterator(null, d, problemQuadratic, 0);
        CompositeIterator compiter = new CompositeIterator(cdi, false);

        System.out.println("*********** NEW Analysis: \n\n");

        int good, bad;
        good=bad=0;
        while (compiter.hasNext()) {
            CompositeIterator.State state = (CompositeIterator.State)compiter.next();
```

```
            DiagramTreeNode dtn = (DiagramTreeNode)state.getLeafIteratorState();
            Diagram diag = dtn.getDiagram();

            if (diag.getBegin() == diag.getEnd() && dtn.getLeaf()) {
                good++;
                System.out.println("NEW diagram = "+diag);
            }
            else {
                bad++;
            }
        }
        System.out.println("Using ComposableIterators, we found "+good+"/"+bad+" good/bad Diagrams.");

        ICancellationDiagramAnalysis analysis =
                CancellationDiagramFactory.instance().newDiagramTree(problem);

        System.out.println("*********** OLD Analysis: \n\n");
        System.out.println(analysis.toString());

        int oldWay = ((CancellationDiagramTree)analysis).getDiagramTreeNodesCount();

        System.out.println("Using ComposableIterators, we found "+good+"/"+bad+" good/bad Diagrams.");
        System.out.println("Using full tree expansion, we found "+oldWay+" Diagrams.");

        //System.out.println("FINAL compiter="+compiter.toString());
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
public class Node {
    private int _id;
    private HashMap _edges;

    Node(int id) {
        _id = id;
        _edges = new HashMap();
    }

    int getID() {
        return _id;
    }

    int degree() {
        return _edges.size();
    }

    private void addEdge(Edge e) {
        Node peer = e.getOpposite(this);
        Edge eprior = (Edge)_edges.get(peer);
        if (eprior == null) {
            _edges.put(peer, e);
        }
        else {
            throw new RuntimeException("Node.addEdge: parallel duplicate edge");
        }
    }

    Edge addEdge(Node peer) {
        Edge e = (Edge)_edges.get(peer);
        if (e == null) {
            e = new Edge(this, peer);
```

```
            _edges.put(peer, e);
            peer.addEdge(e);
        }
        return e;
    }

    Edge getEdge(Node peer) {
        Edge e = (Edge)_edges.get(peer);
        if (e == null) {
            throw new RuntimeException("Node.getEdge: edge not found");
        }
        return e;
    }

    private void delEdge(Edge e) {
        Node peer = e.getOpposite(this);
        Edge eprior = (Edge)_edges.get(peer);
        if (eprior != null) {
            _edges.remove(peer);
        }
        else {
            throw new RuntimeException("Node.delEdge: nonexistent edge");
        }
    }

    Edge delEdge(Node peer) {
        Edge e = (Edge)_edges.get(peer);
        if (e != null) {
            _edges.remove(peer);
            peer.delEdge(e);
        }
        return e;
    }

    Iterator edgeIterator() {
        return _edges.values().iterator();
    }

    public String toString() {
        String s = "";
        s += _id;
        s += "("+_edges.size()+")";
        return s;
    }

    String toStringShort() {
        String s = "";
```

```
        s += _id;
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package cancellation;

import java.util.LinkedList;
import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
public class Path {
    private LinkedList _nodeList;
    private LinkedList _edgeList;
    private Node _srcNode;
    private Edge _srcEdge;
    private Diagram _d;

    Path(Node src, Diagram d) {
        _d = d;
        _srcNode = src;
        _srcEdge = null;
        _edgeList = new LinkedList();
        _nodeList = new LinkedList();
        _nodeList.add(src);
    }

    Path(Edge src, Diagram d) {
        _d = d;
        _srcNode = null;
        _srcEdge = src;
        _nodeList = new LinkedList();
        _edgeList = new LinkedList();
        _edgeList.add(src);
    }

    static Path project(Path p, Diagram d2) {
        Path p2 = new Path(p, d2);
        p2.validate(d2);
        return p2;
    }

    private Path(Path p, Diagram d2) {
        _d = d2;
```

```
        if (p._srcNode == null) _srcNode=null;
        else _srcNode = d2.lookupNode(p._srcNode.getID());


        if (p._srcEdge == null) _srcEdge=null;
        else _srcEdge = d2.lookupEdge(p._srcEdge.getA().getID(), p._srcEdge.getB().getID());

        _edgeList = new LinkedList();
        _nodeList = new LinkedList();

        Iterator itNodes = p._nodeList.iterator();
        for ( ;itNodes.hasNext();) {
            Node nd = (Node)itNodes.next();
            _nodeList.add(d2.lookupNode(nd.getID()));
        }

        Iterator itEdges = p._edgeList.iterator();
        for ( ;itEdges.hasNext();) {
            Edge e = (Edge)itEdges.next();
            _edgeList.add(d2.lookupEdge(e.getA().getID(), e.getB().getID()));
        }
    }

    Iterator iteratorEdges() {
        return _edgeList.iterator();
    }

    boolean isNodePath() {
        return _srcEdge==null;
    }

    Node getSrcNode() {
        if ( ! isNodePath()) {
            throw new RuntimeException("Path.getSrcNode: on a non NodePath");
        }
        return _srcNode;
    }

    public int length() {
        int sz = _nodeList.size() - 1;
        if (isEdgePath()) sz++;
        return sz;
    }

    boolean isEdgePath() {
        return _srcNode==null;
```

```
    }

    Edge getSrcEdge() {
        if ( ! isEdgePath()) {
            throw new RuntimeException("Path.getSrcEdge: on a non EdgePath");
        }
        return _srcEdge;
    }

    void append(Edge e) {
        _d.validateEdge(e);

        _edgeList.add(e);
    }

    void append(Node nd) {
        _d.validateNode(nd);

        _nodeList.add(nd);
    }

    boolean hasEdge(Edge e) {
        _d.validateEdge(e);
        return _edgeList.contains(e);
    }

    int getEdgeIndex(Edge e) {
        _d.validateEdge(e);
        return _edgeList.indexOf(e);
    }

    void cutEdge(Edge e, Node insert) {
        _d.validateEdge(e);
        _d.validateNode(insert);

        if ( ! _edgeList.contains(e)) {
            throw new RuntimeException("Path.cutEdge: nonexistent edge");
        }

        int i = _edgeList.indexOf(e);
        Node a = e.getA();
        Node b = e.getB();
        int ai = _nodeList.indexOf(a);
        int bi = _nodeList.indexOf(b);
        if (ai<0 || bi<0) {
            throw new RuntimeException("Path.cutEdge: nonexistent edge");
        }
```

```
        Node first, second;
        int firsti, secondi;
        if (ai < bi) {
            first = a;
            firsti = ai;
            second = b;
            secondi = bi;
        }
        else if (bi < ai) {
            first = b;
            firsti = bi;
            second = a;
            secondi = ai;
        }
        else {
            throw new RuntimeException("Path.cutEdge: loop edge");
        }

        if (firsti+1 != secondi) {
            throw new RuntimeException("Path.cutEdge: nodeList-edgeList inconsistency");
        }

        _edgeList.remove(e);
        _nodeList.add(secondi, insert);
        // System.out.println("DEBUG inserting "+insert);
        Edge e1 = insert.addEdge(first);
        Edge e2 = insert.addEdge(second);
        _edgeList.add(i, e2);
        _edgeList.add(i, e1);
    }

    public String toString() {
        String s = "[len="+length()+"] ";
        Iterator itNodes = _nodeList.iterator();
        Iterator itEdges = _edgeList.iterator();
        if (this.isNodePath()) {
            Node src = (Node)itNodes.next();
            s += src.toStringShort()+", ";
        }
        else if (this.isEdgePath()) {
            Edge src = (Edge)itEdges.next();
            s += src.toString()+", ";
        }

        for ( ;itNodes.hasNext();) {
            if (this.isNodePath()) {
                Edge e = (Edge)itEdges.next();
```

```
                s += e.toString()+", ";
                Node nd = (Node)itNodes.next();
                s += nd.toStringShort()+", ";
            }
            else if (this.isEdgePath()) {
                Node nd = (Node)itNodes.next();
                s += nd.toStringShort()+", ";
                if (itEdges.hasNext()) {
                    Edge e = (Edge)itEdges.next();
                    s += e.toString()+", ";
                }
            }
        }
        return s;
    }

    private void validate() {
        Iterator itNodes = _nodeList.iterator();
        for ( ;itNodes.hasNext();) {
            Node nd = (Node)itNodes.next();
            _d.validateNode(nd);
        }

        Iterator itEdges = _edgeList.iterator();
        for ( ;itEdges.hasNext();) {
            Edge e = (Edge)itEdges.next();
            _d.validateEdge(e);
        }

        if ((_srcNode != null) && (_srcNode != _nodeList.getFirst())) {
            throw new RuntimeException("Path.validate: _srcNode inconsistency");
        }
        if ((_srcEdge != null) && (_srcEdge != _edgeList.getFirst())) {
            throw new RuntimeException("Path.validate: _srcEdge inconsistency");
        }
    }

    void validate(Diagram d) {
        if (_d != d) {
            throw new RuntimeException("Path.validate: invalid associated Diagram");
        }
        validate();
    }

}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package equation;

import java.util.LinkedList;
import java.util.Iterator;
import letter.Letter;
import letter.LetterFactory;

/**
 *
 * @author grouptheory
 */
public class GroupEquation {

    private LinkedList _letters;

    public GroupEquation(String s) {
        _letters = new LinkedList();
        _letters.addAll(LetterFactory.instance().parse(s));
    }

    public GroupEquation(GroupEquation eq) {
        _letters = new LinkedList();
        _letters.addAll(eq._letters);
    }

    public String toString() {
        String s = "";
        for (Iterator it = _letters.iterator(); it.hasNext(); ) {
            Letter let = (Letter)it.next();
            s += (let.toString());
        }
        return s;
    }

    GroupEquation() {
        _letters = new LinkedList();
    }

    void appendLetter(Letter let) {
        _letters.addLast(let);
    }
```

```
    public int length() {
        return _letters.size();
    }

    public Letter popLetter() {
        return (Letter)_letters.removeFirst();
    }

    public LetterIterator getLetterIterator() {
        LetterIterator letit = new LetterIterator();
        return letit;
    }

    public class LetterIterator {
        private java.util.Iterator _iterator;

        LetterIterator() {
            _iterator = _letters.iterator();
        }

        public Letter next() {
            return (Letter)_iterator.next();
        }

        public Boolean hasNext() {
            return _iterator.hasNext();
        }
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package equation;

import java.util.Iterator;
import java.util.HashMap;
import letter.Letter;

/**
 *
 * @author grouptheory
 */
public class Latex {

    private static Latex _instance;

    private Latex() {
    }

    public static Latex instance() {
        if (_instance == null) {
            _instance = new Latex();
        }
        return _instance;
    }

    public String renderGroupEquation(GroupEquation eq) {
        String s = "";
        for (GroupEquation.LetterIterator it = eq.getLetterIterator(); it.hasNext();) {
            Letter let = it.next();
            s += letter.Latex.instance().render(let);
        }
        s += "=_F 1";
        return s;
    }

    public String renderQuadraticSystem(QuadraticSystem qs) {
        String s = "";
        s += equation.Latex.instance().renderGroupEquation(qs.getEquation());
        HashMap equiv = qs.getEquivalences();
        if (equiv.size()>0) {
            s += " $, where $";
            for (Iterator it=equiv.keySet().iterator(); it.hasNext();) {
                Letter let = (Letter)it.next();
```

```
                Letter leteq = (Letter)equiv.get(let);

                s += letter.Latex.instance().render(let);
                s += "=";
                s += letter.Latex.instance().render(leteq);
                if (it.hasNext()) s+= ", ";
            }
        }
        s += ".";
        return s;
    }


    public String renderQSAsText(QuadraticSystem qs) {
        String s = "{\\bf Quadratic System:}\n";
        s += "$";
        s += renderQuadraticSystem(qs);
        s += "$";
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package equation;

/**
 *
 * @author grouptheory
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

        GroupEquation eqn = new GroupEquation("z1+.z2+.z1-.z2-.c1+.c2+.c1-.c2-.z1+.z2+.z1-.z2-.c1+.c2+.c1-.c2-.");

        QuadraticSystem qs;
        qs = QuadraticSystemFactory.instance().newQuadraticSystem(eqn);

        System.out.println("eqn: "+eqn);
        System.out.println("qs: "+qs);
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package equation;

import java.util.HashMap;
import java.util.LinkedList;
import java.util.Iterator;
import letter.Variable;
import letter.Letter;

/**
 *
 * @author grouptheory
 */
public class QuadraticSystem {
    private GroupEquation _eqn;
    private HashMap _equivalences;
    private HashMap _equivalencesReverse;

    QuadraticSystem() {
        _eqn = new GroupEquation();
        _equivalences = new HashMap();
        _equivalencesReverse = new HashMap();
    }

    void appendLetter(Letter let) {
        _eqn.appendLetter(let);
    }

    void addEquivalence(Variable v1, Variable v2) {
        if (v1 == v2) {
            throw new RuntimeException("QuadraticSystem.addEquivalence: v1 == v2");
        }
        _equivalences.put(v1, v2);
        _equivalencesReverse.put(v2, v1);
    }

    public GroupEquation getEquation() {
        GroupEquation eqn = new GroupEquation(_eqn);
        return eqn;
    }

    public HashMap getEquivalences() {
        HashMap equiv = new HashMap();
```

```
        equiv.putAll(_equivalences);
        equiv.putAll(_equivalencesReverse);
        return equiv;
    }

    public String toString() {
        String s = "";
        s += _eqn.toString() + "=1; \n";
        for (Iterator it=_equivalences.keySet().iterator(); it.hasNext();) {
            Letter let = (Letter)it.next();
            s += (let.toString() + "="+ ((Letter)_equivalences.get(let)).toString() + "; \n");
        }
        s += "\n";
        return s;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package equation;

import letter.Letter;
import letter.Variable;
import letter.LetterFactory;
import java.util.HashMap;
import java.util.HashSet;

/**
 *
 * @author grouptheory
 */
public class QuadraticSystemFactory {

    private static QuadraticSystemFactory _instance;

    private QuadraticSystemFactory() {
    }

    public static QuadraticSystemFactory instance() {
        if (_instance == null) {
            _instance = new QuadraticSystemFactory();
        }
        return _instance;
    }

    public QuadraticSystem newQuadraticSystem(GroupEquation eqn) {
        QuadraticSystem qs = new QuadraticSystem();


        HashSet vars = new HashSet();
        for (GroupEquation.LetterIterator it = eqn.getLetterIterator(); it.hasNext();) {
            Letter let = it.next();
            vars.add(let);
        }

        HashMap letter2count = new HashMap();
        for (GroupEquation.LetterIterator it = eqn.getLetterIterator(); it.hasNext();) {
            Letter let = it.next();

            if ( ! let.isConstant()) {
                Variable var = (Variable)let;
```

```
                int ct = getCount(letter2count, var);
                if (ct==2) {
                    Variable subs = LetterFactory.instance().newUnusedVariable(vars,0,var.isPositive());
                    vars.add(subs);

                    qs.appendLetter(subs);
                    qs.addEquivalence(var, subs);
                }
                else {
                    qs.appendLetter(var);
                    incrementCount(letter2count, var);
                }
            }
            else {
                qs.appendLetter(let);
            }
        }
        return qs;
    }

    private int getCount(HashMap letter2count, Letter let) {
        Letter key;
        if (let.isPositive()) {
            key = let;
        }
        else {
            key = let.getInverse();
        }
        Integer ct = (Integer)letter2count.get(key);
        if (ct==null) return 0;
        else return ct.intValue();
    }


    private void incrementCount(HashMap letter2count, Letter let) {
        Letter key;
        if (let.isPositive()) {
            key = let;
        }
        else {
            key = let.getInverse();
        }
        Integer ct = (Integer)letter2count.get(key);
        if (ct==null) letter2count.put(key, 1);
        else {
            letter2count.put(key, 1+ct.intValue());
        }
```

```
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import utility.AbstractDecorable;
import letter.Letter;

/**
 *
 * @author grouptheory
 */
public class Base extends AbstractDecorable {
    private BaseSet _owner;
    private Boundary _begin;
    private Boundary _end;
    private Letter _label;
    private Base _dual;
    private Constraint _cons;

    Base(Boundary begin, Boundary end, Letter label) {
        if ( begin.getID() > end.getID()) {
            throw new RuntimeException("Base.ctor boundaries out of order");
        }
        _begin = begin;
        _end = end;
        _label = label;
        _dual = null;

        if (this.getLabel().isConstant()) {
            _cons = null;
        }
        else {
            _cons = new Constraint(this);
        }

        _owner = null;
    }

    public Boundary getBegin() {
        return _begin;
    }

    public Boundary getEnd() {
        return _end;
```

```
    }

    public void move(Boundary begin, Boundary end) {
        if (begin==null) {
            throw new RuntimeException("Base.move: begin == null");
        }
        if (end==null) {
            throw new RuntimeException("Base.move: end == null");
        }
        if (_cons.size() > 0) {
            throw new RuntimeException("Base.move: _cons.size() > 0");
        }

        if ( begin.getID() > end.getID()) {
            _label = _label.getInverse();
            Boundary swap = begin;
            begin = end;
            end = swap;
        }
        _begin = begin;
        _end = end;
    }

    public boolean isEmpty() {
        return this.getBegin() == this.getEnd();
    }

    boolean isConstant() {
        return this.getLabel().isConstant();
    }

    public Letter getLabel() {
        return _label;
    }

    boolean contains(Boundary b) {
        if (_begin==null) {
            throw new RuntimeException("Base.contains: _begin == null");
        }
        if (_end==null) {
            throw new RuntimeException("Base.contains: _end == null");
        }
        if (b==null) {
            throw new RuntimeException("Base.contains: b == null");
        }

        return (b.getID() >= _begin.getID() &&
```

```
                b.getID() <= _end.getID());
    }

    public Constraint getConstraint() {
        if (this.getLabel().isConstant()) {
            throw new RuntimeException("Base.getConstraint: constant base has no constraint");
        }
        return _cons;
    }

    void _setConstraint(Constraint c) {
        if (this.getLabel().isConstant()) {
            throw new RuntimeException("Base.getConstraint: constant base has no constraint");
        }
        _cons = c;
    }

    public Base getDual() {
        if (this.getLabel().isConstant()) {
            throw new RuntimeException("Base.getDual: constant base has no dual");
        }
        return _dual;
    }

    void setDual(Base dual) {
        if (this.getDual() != null) {
            throw new RuntimeException("Base.setDual: base already has dual ");
        }
        if (this.getLabel().isConstant()) {
            throw new RuntimeException("Base.setDual: constant base has no dual");
        }
        _dual = dual;
    }

    BaseSet getOwner() {
        return _owner;
    }

    void _setOwner(BaseSet owner) {
        if (this.getOwner() != null && owner!=null) {
            throw new RuntimeException("Base._setOwner: base is already owned");
        }
        _owner = owner;
    }

    public String toString() {
        String s = "";
```

```
        s += "[";
        s += _begin.toString();
        s += "-";
        s += _end.toString();
        s += ":";
        s += _label.toString();
        s += "] ";
        s += this.hashCode();
        if ( ! this.isConstant()) {
            s += " >> ";
            s += this.getDual().hashCode();
            s += " :: ";
            s += _cons.toString();
        }
        return s;
    }


    public String toStringShort() {
        String s = "";
        s += "[";
        s += _begin.toString();
        s += "-";
        s += _end.toString();
        s += ":";
        s += _label.toString();
        s += "] ";
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import java.util.Comparator;

/**
 *
 * @author grouptheory
 */
public class BaseComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        Base b1 = (Base)o1;
        Base b2 = (Base)o2;

        if (b1.getOwner().getOwner() != b2.getOwner().getOwner()) {
            throw new RuntimeException("BaseComparator.compareTo: incomparable GEs");
        }

        // carrier must be first
        // nonempty variable with the smallest left boundary (and longest length in case of ties)

        // nonempty < empty
        if (!b1.isEmpty() && b2.isEmpty()) {
            return -1;
        }
        else if (b1.isEmpty() && !b2.isEmpty()) {
            return +1;
        }
        else {
            // variables < constants
            if (!b1.isConstant() && b2.isConstant()) {
                return -1;
            }
            else if (b1.isConstant() && !b2.isConstant()) {
                return +1;
            }
            else {
                // small left boundary < big left boundary
                 if (b1.getBegin().getID() < b2.getBegin().getID()) {
                    return -1;
                }
                else if (b1.getBegin().getID() > b2.getBegin().getID()) {
                    return +1;
```

```
                }
                else {
                    // longer < shorter
                    int len1 = b1.getEnd().getID() - b1.getBegin().getID();
                    int len2 = b2.getEnd().getID() - b2.getBegin().getID();
                    if (len1 > len2) {
                        return -1;
                    }
                    else if (len1 < len2) {
                        return +1;
                    }
                    else {
                        // earlier label < later label
                        if (b1.getLabel().getID() < b2.getLabel().getID()) {
                            return -1;
                        }
                        else if (b1.getLabel().getID() > b2.getLabel().getID()) {
                            return +1;
                        }
                        else {
                            // smaller hashcode < greater hashcode
                            if (b1.hashCode() < b2.hashCode()) {
                                return -1;
                            }
                            else if (b1.hashCode() > b2.hashCode()) {
                                return +1;
                            }
                            else {
                                // equal!
                                return 0;
                            }
                        }
                    }
                }
            }
        }
    }

    public boolean equals(Object obj) {
        return obj==this;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import utility.AbstractDecorator;

/**
 *
 * @author grouptheory
 */
public class BaseDecorator extends AbstractDecorator {

    public BaseDecorator() {
    }

    public Base getBase() {
        return (Base)super.getOwner();
    }

    public String getName() {
        return getOwner().lookupDecoratorName(this);
    }

    public void attach(String name, Base owner) {
        setOwner(owner);
        getOwner().attachDecorator(name, this);
    }

    public void detach() {
        getOwner().detachDecorator(this);
        setOwner(null);
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

/**
 *
 * @author grouptheory
 */
public class BaseLayoutDecorator extends BaseDecorator {

    static final String NAME = "Layout";

    private double _y;
    private double _x1;
    private double _x2;

    BaseLayoutDecorator(double x1, double x2, double y) {
        _x1 = x1;
        _x2 = x2;
        _y = y;
    }

    double getX1() {
        return _x1;
    }

    double getX2() {
        return _x2;
    }

    double getY() {
        return _y;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
public class BaseLayoutDecoratorFactory {
    private int[] _occ;
    private boolean[][] _used;
    private int _slots;
    private int _bd;
    private int _bs;

    private double _width;
    private double _height;
    private double _boundaryspace;
    private double _basespace;

    private double WIDTH;
    private double HEIGHT;

    static double applyToAllBases(GE geq, double width, double height) {
        BaseLayoutDecoratorFactory bldf = new BaseLayoutDecoratorFactory(geq, width, height);
        for (Iterator it=geq.iteratorBases(); it.hasNext();) {
            Base bs=(Base)it.next();
            BaseLayoutDecorator bld = bldf.newBaseHeightDecorator(bs);
            bld.attach(BaseLayoutDecorator.NAME, bs);
        }
        return bldf._boundaryspace;
    }

    private BaseLayoutDecoratorFactory(GE geq, double width, double height) {
        _bd = geq.getNumberOfBoundaries();
        _bs = geq.getNumberOfBases();

        _occ = new int[_bd];
        for (int i=0; i<_bd; i++) {
            _occ[i]=0;
        }
```

```
        for (Iterator it=geq.iteratorBases(); it.hasNext();) {
            Base bs=(Base)it.next();
            this.incorporate(bs);
        }

        initSlots();

        HEIGHT = height;
        WIDTH = width;
        _basespace = HEIGHT/(double)this.maxOcc();
        _boundaryspace = WIDTH/((double)_bd-1);
    }

    private void incorporate(Base bs) {
        int left = bs.getBegin().getID();
        int right = bs.getEnd().getID();
        for (int i=left; i<=right; i++) {
            _occ[i]++;
        }
    }

    private int maxOcc() {
        int max=0;
        for (int i=0; i<_bd; i++) {
            if (_occ[i]>max) max=_occ[i];
        }
        return max+2;
    }

    private void initSlots() {
        _slots = maxOcc();
        _used = new boolean[_bd][_slots+1];

        for (int i=0; i<_bd; i++) {
            for (int j=0; j<_slots; j++) {
                _used[i][j]=false;
            }
        }
    }

    private BaseLayoutDecorator newBaseHeightDecorator(Base bs) {

        boolean success = false;
        int height=0;
        int left = bs.getBegin().getID();
        int right = bs.getEnd().getID();
        for (int h=1; h<=_slots; h++) {
```

```
            if (isFree(left, right, h)) {
                height = h;
                success = true;
                break;
            }
        }
        if (success) {
            markUsed(left, right, height);
            return new BaseLayoutDecorator(left*_boundaryspace,
                                           right*_boundaryspace,
                                           height*_basespace);
        }
        else {
            throw new RuntimeException("DiagramAllocator.assignHeight: failed for bs="+bs);
        }
    }

    private boolean isFree(int left, int right, int h) {
        for (int i=left;i<=right;i++) {
            if (_used[i][h]) return false;
        }
        return true;
    }

    private void markUsed(int left, int right, int h) {
        for (int i=left;i<=right;i++) {
            _used[i][h] = true;
        }
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import java.util.LinkedList;
import java.util.Iterator;
import java.util.HashMap;
import java.util.TreeMap;
import java.util.TreeSet;
import letter.Letter;

/**
 *
 * @author grouptheory
 */
public class BaseSet {
    private GE _owner;
    private LinkedList _bsList;

    BaseSet(GE owner) {
        _owner = owner;
        _bsList = new LinkedList();
    }

    GE getOwner() {
        return _owner;
    }

    BaseSet duplicate(GE owner, TreeMap old2new_bdmap, HashMap old2new_bsmap) {
        old2new_bsmap.clear();
        BaseSet bs2 = new BaseSet(owner);
        for (Iterator it = this.iterator(); it.hasNext();) {
            Base b = (Base)it.next();
            Boundary begin2 = (Boundary)old2new_bdmap.get(b.getBegin());
            Boundary end2 = (Boundary)old2new_bdmap.get(b.getEnd());
            Letter let = b.getLabel();
            Base b2 = new Base(begin2, end2, let);
            bs2.add(b2);
            old2new_bsmap.put(b, b2);
        }

        for (Iterator it = this.iterator(); it.hasNext();) {
            Base b = (Base)it.next();
            if (b.isConstant()) continue;
```

```
            Base bDual = b.getDual();

            Base b2 = (Base)old2new_bsmap.get(b);
            Base bDual2 = (Base)old2new_bsmap.get(bDual);

            b2.setDual(bDual2);
        }

        for (Iterator it = this.iterator(); it.hasNext();) {
            Base b = (Base)it.next();
            if (b.isConstant()) continue;

            Constraint c = b.getConstraint();
            Constraint c2 = c.duplicate(old2new_bdmap, old2new_bsmap);

            Base b2 = (Base)old2new_bsmap.get(b);
            b2._setConstraint(c2);
        }

        return bs2;
    }

    void add(Base b) {
        _bsList.addLast(b);
        b._setOwner(this);
    }

    void remove(Base b) {
        if ( ! _bsList.contains(b)) {
            throw new RuntimeException("BaseSet.remove: base not present");
        }
        b._setOwner(null);
        _bsList.remove(b);
    }

    Iterator iterator() {
        return _bsList.iterator();
    }

    int getNumberOfBases() {
        return _bsList.size();
    }

    public String toString() {
        String s = "";
        s += "Bases: {\n";
```

```
        TreeSet ts = new TreeSet(new BaseComparator());
        ts.addAll(_bsList);

        for (Iterator it=ts.iterator(); it.hasNext();) {
            Base bs=(Base)it.next();
            s += "        ";
            s += bs.toString();
            s += "\n";
        }
        s += "       }";
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

/**
 *
 * @author grouptheory
 */
public class Boundary implements java.lang.Comparable {
    private BoundarySet _owner;

    Boundary() {
        _owner = null;
    }

    public int getID() {
        return _owner.getIndex(this);
    }

    BoundarySet getOwner() {
        return _owner;
    }

    void _setOwner(BoundarySet owner) {
        if (this.getOwner() != null && owner!=null) {
            throw new RuntimeException("Boundary._setOwner: boundary is already owned");
        }
        _owner = owner;
    }

    public String toString() {
        String s = "";
        s += ""+getID();
        return s;
    }

    public int compareTo(Object o) {
        if (this.getOwner() == null) {
            throw new RuntimeException("Boundary.compareTo: _owner = null");
        }
        if (! (o instanceof Boundary)) {
            throw new RuntimeException("Boundary.compareTo: bad type");
        }
        Boundary obd = (Boundary)o;
```

```
        if (obd.getOwner() == null) {
            throw new RuntimeException("Boundary.compareTo: obd = null");
        }
        if (this.getOwner() != obd.getOwner()) {
            throw new RuntimeException("Boundary.compareTo: obd and this are in different GEs");
        }

        if (this.getID() < obd.getID()) return -1;
        else if(this.getID() > obd.getID()) return +1;
        else return 0;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import java.util.LinkedList;
import java.util.TreeMap;
import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
public class BoundarySet {
    private GE _owner;
    private LinkedList _bdList;

    BoundarySet(GE owner) {
        this(owner,0);
    }

    BoundarySet(GE owner, int n) {
        _owner = owner;
        _bdList = new LinkedList();
        for (int i=0; i<n; i++) {
            Boundary b = new Boundary();
            this.add(b);
        }
    }

    GE getOwner() {
        return _owner;
    }

    BoundarySet duplicate(GE owner, TreeMap old2new_bdmap) {
        old2new_bdmap.clear();
        BoundarySet bd2 = new BoundarySet(owner);
        for (Iterator it = this.iterator(); it.hasNext();) {
            Boundary b = (Boundary)it.next();
            Boundary b2 = new Boundary();
            bd2.add(b2);
            old2new_bdmap.put(b, b2);
        }
        return bd2;
    }
```

```
    int getIndex(Boundary b) {
        if ( ! _bdList.contains(b)) {
            throw new RuntimeException("BoundarySet.getIndex: bad boundary");
        }
        int idx = _bdList.indexOf(b);
        return idx;
    }

    void add(Boundary b) {
        if (_bdList.contains(b)) {
            throw new RuntimeException("BoundarySet.add: duplicate boundary");
        }
        _bdList.addLast(b);
        b._setOwner(this);
    }

    void remove(Boundary b) {
        if ( ! _bdList.contains(b)) {
            throw new RuntimeException("BoundarySet.remove: nonexistent boundary");
        }
        b._setOwner(null);
        _bdList.remove(b);
    }

    Iterator iterator() {
        return _bdList.iterator();
    }

    public Boundary appendNewBoundary() {
        Boundary bd = new Boundary();
        this.add(bd);
        return bd;
    }

    public Boundary insertNewBoundaryAfter(Boundary b) {
        int idx = getIndex(b);
        Boundary newb = new Boundary();
        _bdList.add(idx+1, newb);
        newb._setOwner(this);
        return newb;
    }

    public Boundary insertNewBoundaryBefore(Boundary b) {
        int idx = getIndex(b);
        Boundary newb = new Boundary();
        _bdList.add(idx-1, newb);
```

```
        newb._setOwner(this);
        return newb;
    }

    public Boundary getFirst() {
        return (Boundary)_bdList.getFirst();
    }

    Boundary getNth(int i) {
        return (Boundary)_bdList.get(i);
    }

    public Boundary getLast() {
        return (Boundary)_bdList.getLast();
    }

    public int getNumberOfBoundaries() {
        return _bdList.size();
    }

    public Boundary nextBoundary(Boundary bd) {
        int idx = _bdList.indexOf(bd);
        if (idx == _bdList.size()-1) {
            return null;
        }
        else {
            return (Boundary)_bdList.get(idx+1);
        }
    }

    public Boundary prevBoundary(Boundary bd) {
        int idx = _bdList.indexOf(bd);
        if (idx == 0) {
            return null;
        }
        else {
            return (Boundary)_bdList.get(idx-1);
        }
    }

    public String toString() {
        String s = "";
        s += "Boundaries: {\n";
        for (Iterator it=this.iterator(); it.hasNext();) {
            Boundary bd=(Boundary)it.next();
            s += "             ";
            s += bd.toString();
```

```
            s += " ";
            s += bd.hashCode();
            s += "\n";
        }
        s += "}";
        return s;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import cancellation.*;
import equation.GroupEquation;
import equation.QuadraticSystem;
import ge.GE;
import ge.GEFactory;

/**
 *
 * @author grouptheory
 */
public class CancellationDiagramAnalysis_GEDecorator
        implements ICancellationDiagramAnalysisDecorator {

    public CancellationDiagramAnalysis_GEDecorator() {
    }

    public String texify(ICancellationDiagramAnalysis analysis, DiagramTreeNode dtn) {

        QuadraticSystem qs = analysis.getQuadraticSystem();
        Diagram d = dtn.getDiagram();

        GEFactory gef = GEFactory.instance();
        GE geq = gef.newGE(d, qs);

        String s="";
        if (params.MKParams.FLAG_REPORT_GE_STRUCTURES) {
            s += ge.Latex.instance().renderGEasText(geq);
        }
        if (params.MKParams.FLAG_REPORT_GE_PICTURES) {
            s += ge.Latex.instance().renderGEasGraphics(geq);
        }

        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import java.util.TreeMap;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import letter.Letter;
import letter.Variable;

/**
 *
 * @author grouptheory
 */
public class Constraint {
    private Base _bs;
    private TreeMap _bd2bd;

    Constraint(Base b) {
        _bs = b;
        _bd2bd = new TreeMap();
    }

    void initialize() {
        Base dual = _bs.getDual();
        Variable v = (Variable)_bs.getLabel();
        Variable vdual = (Variable)dual.getLabel();

        boolean flipDualBoundaries = false;
        if ((v.isPositive() && !vdual.isPositive()) ||
            (!v.isPositive() && vdual.isPositive())) {
            flipDualBoundaries = true;
        }

        if (!flipDualBoundaries) {
            this.add(_bs.getBegin(), dual.getBegin());
            this.add(_bs.getEnd(), dual.getEnd());
        }
        else {
            this.add(_bs.getBegin(), dual.getEnd());
            this.add(_bs.getEnd(), dual.getBegin());
        }
    }
```

```
    private Constraint(Base b, TreeMap bd2bd2) {
        this(b);
        _bd2bd.putAll(bd2bd2);
    }

    Constraint duplicate(TreeMap old2new_bdmap, HashMap old2new_bsmap) {
        Base bs2 = (Base)old2new_bsmap.get(_bs);
        TreeMap bd2bd2 = new TreeMap();

        for (Iterator it=_bd2bd.entrySet().iterator(); it.hasNext();) {
            Map.Entry ent = (Map.Entry)it.next();
            Boundary b = (Boundary)ent.getKey();
            Boundary bDual = (Boundary)ent.getValue();

            Boundary b2 = (Boundary)old2new_bdmap.get(b);
            Boundary bDual2 = (Boundary)old2new_bdmap.get(bDual);
            bd2bd2.put(b2, bDual2);
        }
        return new Constraint(bs2, bd2bd2);
    }

    void add(Boundary b, Boundary bdual) {
        if ( ! _bs.contains(b)) {
            throw new RuntimeException("Constraint.add: boundary not in base");
        }
        if ( ! _bs.getDual().contains(bdual)) {
            throw new RuntimeException("Constraint.add: dual boundary not in dual base");
        }
        if ( _bd2bd.containsKey(b) ) {
            throw new RuntimeException("Constraint.add: constraint already exists");
        }
        _bd2bd.put(b, bdual);
    }

    void remove(Boundary b, Boundary bdual) {
        if ( ! _bs.contains(b)) {
            throw new RuntimeException("Constraint.remove: boundary "+b+" is not in base "+_bs);
        }
        if ( ! _bs.getDual().contains(bdual)) {
            throw new RuntimeException("Constraint.remove: dual boundary not in dual base");
        }
        if ( ! _bd2bd.containsKey(b) ) {
            throw new RuntimeException("Constraint.remove: constraint doesn’t exist");
        }
        _bd2bd.remove(b);
    }
```

```
    public Boundary getDual(Boundary b) {
        if ( ! _bd2bd.containsKey(b) ) {
            throw new RuntimeException("Constraint.getDual: constraint doesn’t exist");
        }
        return (Boundary)_bd2bd.get(b);
    }

    Base getBase() {
        return _bs;
    }

    public Iterator iteratorBoundary() {
        return _bd2bd.keySet().iterator();
    }

    public int size() {
        return _bd2bd.size();
    }

    public String toString() {
        String s = "";
        for (Iterator it=this.iteratorBoundary(); it.hasNext();) {
            Boundary bd=(Boundary)it.next();
            Boundary bdDual = this.getDual(bd);
            s += bd.toString();
            s += "->";
            s += bdDual.toString();
            if (it.hasNext()) s += ", ";
        }
        return s;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import utility.AbstractDecorable;
import java.util.TreeMap;
import java.util.HashMap;
import java.util.Iterator;
import letter.Constant;
import letter.Variable;

/**
 *
 * @author grouptheory
 */
public class GE extends AbstractDecorable {
    private BoundarySet _bdSet;
    private BaseSet _bsSet;

    GE() {
        setBoundarySet(new BoundarySet(this));
        setBaseSet(new BaseSet(this));

        appendNewBoundary();
    }

    private void setBoundarySet(BoundarySet bdSet) {
        _bdSet = bdSet;
    }

    private void setBaseSet(BaseSet bsSet) {
        _bsSet = bsSet;
    }

    public GE duplicate() {
        TreeMap old2new_bdmap = new TreeMap();
        HashMap old2new_bsmap = new HashMap();
        return duplicate(old2new_bdmap, old2new_bsmap);
    }


    public GE duplicate(TreeMap old2new_bdmap, HashMap old2new_bsmap) {
        GE ge2 = new GE();
        BoundarySet bd2 = this._bdSet.duplicate(ge2, old2new_bdmap);
```

```
        ge2.setBoundarySet(bd2);

        BaseSet bs2 = this._bsSet.duplicate(ge2, old2new_bdmap, old2new_bsmap);
        ge2.setBaseSet(bs2);

        return ge2;
    }

    public Iterator iteratorBoundaries() {
        return _bdSet.iterator();
    }

    public Boundary appendNewBoundary() {
        return _bdSet.appendNewBoundary();
    }

    public Boundary insertNewBoundaryAfter(Boundary b) {
        return _bdSet.insertNewBoundaryAfter(b);
    }

    public Boundary insertNewBoundaryBefore(Boundary b) {
        return _bdSet.insertNewBoundaryBefore(b);
    }

    public int getNumberOfBoundaries() {
        return _bdSet.getNumberOfBoundaries();
    }

    public boolean isUseless(Boundary b) {
        int index = _bdSet.getIndex(b);

        for (Iterator it = this.iteratorBases(); it.hasNext();) {
            Base bs = (Base)it.next();

            if (bs.getBegin() == b) return false;
            if (bs.getEnd() == b) return false;

            if (bs.getLabel().isConstant()) continue;

            Constraint c = bs.getConstraint();
            for (Iterator itcon=c.iteratorBoundary(); itcon.hasNext();) {
                Boundary bdused = (Boundary)itcon.next();
                if (bdused == b) {
                    return false;
                }
            }
        }
```

```
        return true;
    }

    public void removeBoundary(Boundary b) {
        _bdSet.remove(b);
    }

    public Boundary getFirstBoundary() {
        return _bdSet.getFirst();
    }

    public Boundary getNthBoundary(int i) {
        return _bdSet.getNth(i);
    }

    public Boundary getLastBoundary() {
        return _bdSet.getLast();
    }

    public Boundary nextBoundary(Boundary bd) {
        return _bdSet.nextBoundary(bd);
    }

    public Boundary prevBoundary(Boundary bd) {
        return _bdSet.prevBoundary(bd);
    }

    public Base addNewConstantBase(Boundary begin, Constant c) {
        Boundary end = _bdSet.nextBoundary(begin);
        if (end==null) {
            end = _bdSet.appendNewBoundary();
        }
        Base bs = new Base(begin, end, c);
        _bsSet.add(bs);
        return bs;
    }

    public Base addNewVariableBase(Boundary begin, Boundary end, Variable v,
                                   Boundary beginDual, Boundary endDual) {
        Base bs = new Base(begin, end, v);

        Variable vDual = v;
        if (beginDual.getID() > endDual.getID()) {
            vDual = (Variable)v.getInverse();
            Boundary swap = beginDual;
            beginDual = endDual;
            endDual = swap;
```

```
        }
        Base bsDual = new Base(beginDual, endDual, vDual);

        bs.setDual(bsDual);
        bsDual.setDual(bs);

        _bsSet.add(bs);
        _bsSet.add(bsDual);

        bs.getConstraint().initialize();
        bsDual.getConstraint().initialize();

        return bs;
    }

    public Iterator iteratorBases() {
        return _bsSet.iterator();
    }

    public int getNumberOfBases() {
        return _bsSet.getNumberOfBases();
    }

    public void addNewConstraint(Base bs, Boundary bd, Boundary bdDual) {
        if (bs==null) {
            throw new RuntimeException("GE.addNewConstraint: bs == null");
        }
        if (bd==null) {
            throw new RuntimeException("GE.addNewConstraint: bd == null");
        }
        if (bdDual==null) {
            throw new RuntimeException("GE.addNewConstraint: bdDual == null");
        }
        if (bs.getOwner()!=_bsSet) {
            throw new RuntimeException("GE.addNewConstraint: bs is bad");
        }
        if (bd.getOwner()!=_bdSet) {
            throw new RuntimeException("GE.addNewConstraint: bd is bad");
        }
        if (bdDual.getOwner()!=_bdSet) {
            throw new RuntimeException("GE.addNewConstraint: bdDual is bad");
        }
        Constraint c = bs.getConstraint();
        c.add(bd, bdDual);

        Base bsDual = bs.getDual();
        Constraint cDual = bsDual.getConstraint();
```

```
        cDual.add(bdDual, bd);
    }

    public void removeConstraint(Base bs, Boundary bd, Boundary bdDual) {
        if (bs==null) {
            throw new RuntimeException("GE.removeConstraint: bs == null");
        }
        if (bd==null) {
            throw new RuntimeException("GE.removeConstraint: bd == null");
        }
        if (bdDual==null) {
            throw new RuntimeException("GE.removeConstraint: bdDual == null");
        }
        Constraint c = bs.getConstraint();
        c.remove(bd, bdDual);

        Base bsDual = bs.getDual();
        Constraint cDual = bsDual.getConstraint();
        cDual.remove(bdDual, bd);
    }

    public void removeBase(Base bs) {
        if (bs==null) {
            throw new RuntimeException("GE.removeBase: bs == null");
        }
        _bsSet.remove(bs);
    }

    public void collapseBase(Base bs) {
        if (bs==null) {
            throw new RuntimeException("GE.collapseBase: bs == null");
        }
        if (bs.getLabel().isConstant()) {
            throw new RuntimeException("GE.collapseBase: bs == constant");
        }
        Base dual = bs.getDual();
        if (!dual.isEmpty() &&
           ((bs.getBegin() != dual.getBegin()) ||
            (bs.getEnd() != dual.getEnd()) ||
            (bs.getLabel() != dual.getLabel()))) {
            throw new RuntimeException("GE.collapseBase: bs not collapsible");
        }

        Constraint c = bs.getConstraint();
        do {
            if (c.size() > 0) {
                Iterator it = c.iteratorBoundary();
```

```
                Boundary bd = (Boundary)it.next();
                Boundary bdDual = c.getDual(bd);
                removeConstraint(bs,bd, bdDual);
            }
        }
        while (c.size() > 0);

        bs.move(bs.getEnd(), bs.getEnd());
    }

    public String toString() {
        String s = "";
        s += _bdSet.toString();
        s += "\n";
        s += _bsSet.toString();
        s += "\n";
        return s;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import utility.AbstractDecorator;

/**
 *
 * @author grouptheory
 */
public class GEDecorator extends AbstractDecorator {

    public GEDecorator() {
    }

    public GE getGE() {
        return (GE)super.getOwner();
    }

    public String getName() {
        return getOwner().lookupDecoratorName(this);
    }

    public void attach(String name, GE owner) {
        setOwner(owner);
        getOwner().attachDecorator(name, this);
    }

    public void detach() {
        getOwner().detachDecorator(this);
        setOwner(null);
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import java.util.Iterator;
import java.util.LinkedList;

/**
 *
 * @author grouptheory
 */
public class GEDegeneracyTester implements IDegeneracyTestLog {


    class Evidence {
        private String _s;

        Evidence(String s) {
            _s = s;
        }

        public String toString() {
            return _s;
        }
    }

    private LinkedList _evidence;

    public void reportEvidence(String s) {
        _evidence.addLast(new Evidence(s));
    }

    private LinkedList _conditions;
    private void AddCondition(IGEDegeneracyCondition cond) {
        _conditions.addLast(cond);
    }


    private boolean _isDegenerate;

    public GEDegeneracyTester(GE geq) {
        _conditions = new LinkedList();
        AddCondition(new GEDegeneracy_ConditionA());
        AddCondition(new GEDegeneracy_ConditionB());
```

```
        AddCondition(new GEDegeneracy_ConditionC());
        AddCondition(new GEDegeneracy_ConditionD());
        AddCondition(new GEDegeneracy_ConditionE());

        _evidence = new LinkedList();
        _isDegenerate = compute(geq);
    }

    public boolean isDegenerate() {
        return _isDegenerate;
    }

    private boolean compute(GE geq) {
        for (Iterator it = _conditions.iterator(); it.hasNext();) {
            IGEDegeneracyCondition cond = (IGEDegeneracyCondition)it.next();
            if (cond.test(geq, this)) {
                return true;
            }
        }
        return false;
    }

    public String toString() {
        String s="";
        for (Iterator it = _evidence.iterator(); it.hasNext();) {
            Evidence ev=(Evidence)it.next();
            s+=(""+ev);
        }
        return s;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
class GEDegeneracy_ConditionA implements IGEDegeneracyCondition {

    public boolean test(GE geq, IDegeneracyTestLog log) {
        // MK: if e(mu)=-e(delta(mu) then mu and delta(mu) cannot intersect
        boolean answer = false;

        for (Iterator it=geq.iteratorBases(); it.hasNext();) {
            Base bs = (Base)it.next();
            if (bs.isConstant()) continue;

            Base bsDual = bs.getDual();
            boolean pos = bs.getLabel().isPositive();
            boolean posDual = bsDual.getLabel().isPositive();
            if ((pos && !posDual) || (!pos && posDual)) {
                Boundary begin1 = bs.getBegin();
                Boundary end1 = bs.getEnd();
                Boundary begin2 = bsDual.getBegin();
                Boundary end2 = bsDual.getEnd();
                boolean intersects =
                        ((begin1.getID() >= begin2.getID() && begin1.getID() < end2.getID()) ||
                        (end1.getID() > begin2.getID() && end1.getID() <= end2.getID()) ||
                        (begin1.getID() <= begin2.getID() && end1.getID() >= end2.getID()) ||
                        (begin2.getID() <= begin1.getID() && end2.getID() >= end1.getID()));
                if (intersects) {
                    String s = "The base "+bs.toStringShort()+" and its dual are of opposite polarity, yet intersect.  ";
                    log.reportEvidence(s);
                    answer = true;
                }
            }
        }
        return answer;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
class GEDegeneracy_ConditionB implements IGEDegeneracyCondition {

    public boolean test(GE geq, IDegeneracyTestLog log) {
        // BK: if e(mu)=e(delta(mu) then mu and delta(mu) do not contain each other properly
        boolean answer = false;
        for (Iterator it=geq.iteratorBases(); it.hasNext();) {
            Base bs = (Base)it.next();
            if (bs.isConstant()) continue;

            Base bsDual = bs.getDual();
            boolean pos = bs.getLabel().isPositive();
            boolean posDual = bsDual.getLabel().isPositive();
            if ((pos && !posDual) || (!pos && posDual)) {
                Boundary begin1 = bs.getBegin();
                Boundary end1 = bs.getEnd();
                Boundary begin2 = bsDual.getBegin();
                Boundary end2 = bsDual.getEnd();
                boolean containsProperly =
                        (begin1.getID() <  begin2.getID() && end1.getID() >= end2.getID()) ||
                        (begin2.getID() <  begin1.getID() && end2.getID() >= end1.getID()) ||
                        (begin1.getID() <= begin2.getID() && end1.getID() >  end2.getID()) ||
                        (begin2.getID() <= begin1.getID() && end2.getID() >  end1.getID());
                if (containsProperly) {
                    String s = "The base "+bs.toStringShort()+" and its dual are of the same polarity, yet one properly co
ntains the other.  ";
                    log.reportEvidence(s);
                    answer = true;
                }
            }
        }
        return answer;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
class GEDegeneracy_ConditionC implements IGEDegeneracyCondition {

    public boolean test(GE geq, IDegeneracyTestLog log) {
        return test1(geq, log) || test2(geq, log);
    }

    private boolean test1(GE geq, IDegeneracyTestLog log) {
        // MK: if two boundary equations...
        boolean answer = false;

        for (Iterator it=geq.iteratorBases(); it.hasNext();) {
            Base bs = (Base)it.next();
            if (bs.isConstant()) continue;

            Base bsDual = bs.getDual();
            boolean pos = bs.getLabel().isPositive();
            boolean posDual = bsDual.getLabel().isPositive();
            int epsiepsi = +1;
            if ((pos && !posDual) || (!pos && posDual)) {
                epsiepsi = -1;
            }

            Boundary b1_prev = null;
            Boundary b1Dual_prev = null;
            Constraint c = bs.getConstraint();
            for (Iterator itcon=c.iteratorBoundary(); itcon.hasNext();) {
                Boundary b1 = (Boundary)itcon.next();
                Boundary b1Dual = c.getDual(b1);

                if (b1_prev==null || b1Dual_prev==null) continue;

                int delta1;
                if (b1.getID() < b1_prev.getID()) {
                    delta1 = -1;
```

```
                }
                else if (b1.getID() > b1_prev.getID()) {
                    delta1 = +1;
                }
                else {
                    delta1 = 0;
                }

                int delta2;
                if (b1Dual.getID() < b1Dual_prev.getID()) {
                    delta2 = -1;
                }
                else if (b1Dual.getID() > b1Dual_prev.getID()) {
                    delta2 = +1;
                }
                else {
                    delta2 = 0;
                }

                if (delta1*delta2 != epsiepsi) {
                    String s = "The polarity of "+bs.toStringShort()+" and its dual contradict the order of some of its bo
undary equations.  ";
                    log.reportEvidence(s);
                    answer = true;
                }

                b1_prev = b1;
                b1Dual_prev = b1Dual;
            }
        }
        return answer;
    }

    private boolean test2(GE geq, IDegeneracyTestLog log) {
        // MK: for a matched pair of bases...
        boolean answer = false;

        for (Iterator it=geq.iteratorBases(); it.hasNext();) {
            Base bs = (Base)it.next();
            if (bs.isConstant()) continue;

            Base bsDual = bs.getDual();

            if (bs.getBegin() == bsDual.getBegin()) {
                Constraint c = bs.getConstraint();
                for (Iterator itcon=c.iteratorBoundary(); itcon.hasNext();) {
                    Boundary b1 = (Boundary)itcon.next();
```

```
                    Boundary b1Dual = c.getDual(b1);

                    if (b1.getID() != b1Dual.getID()) {
                        String s = "The "+bs.toStringShort()+" is matched with its dual, yet their boundary equations do n
ot align.  ";
                        log.reportEvidence(s);
                        answer = true;
                    }
                }
            }
        }
        return answer;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
class GEDegeneracy_ConditionD implements IGEDegeneracyCondition {

    public boolean test(GE geq, IDegeneracyTestLog log) {
        return test1(geq, log) || test2(geq, log);
    }

    private boolean test1(GE geq, IDegeneracyTestLog log) {
        // two distinct constants cannot overlap
        boolean answer = false;

        for (Iterator it=geq.iteratorBases(); it.hasNext();) {
            Base bs = (Base)it.next();
            if ( ! bs.isConstant()) continue;

            for (Iterator it2=geq.iteratorBases(); it2.hasNext();) {
                Base bs2 = (Base)it2.next();
                if ( ! bs2.isConstant()) continue;

                Boundary bd = bs.getBegin();
                Boundary bd2 = bs2.getBegin();
                if (bd != bd2) continue;

                if (bs.getLabel() == bs2.getLabel())
                    continue;

                String s = "Two distinct constants appear at boundary "+bd+".  ";
                log.reportEvidence(s);
                answer = true;
            }
        }
        return answer;
    }
```

```
    private boolean test2(GE geq, IDegeneracyTestLog log) {
        // MK: a variable cannot occur in two distinct coefficient equations
        boolean answer = false;

        for (Iterator vit=geq.iteratorBases(); vit.hasNext();) {
            Base vbs = (Base)vit.next();
            if (vbs.isConstant()) continue;

            for (Iterator cit=geq.iteratorBases(); cit.hasNext();) {
                Base cbs = (Base)cit.next();
                if ( ! cbs.isConstant()) continue;

                Boundary vbd = vbs.getBegin();
                Boundary cbd = cbs.getBegin();

                if (vbd == cbd) {

                    Base vbs2 = (Base)vbs.getDual();

                    for (Iterator cit2=geq.iteratorBases(); cit2.hasNext();) {
                        Base cbs2 = (Base)cit2.next();
                        if ( ! cbs2.isConstant()) continue;

                        Boundary vbd2 = vbs2.getBegin();
                        Boundary cbd2 = cbs2.getBegin();
                        Boundary vbd2end = vbs2.getEnd();
                        Boundary cbd2end = cbs2.getEnd();

                        if ((vbd2 == cbd) && (vbd2end == cbd2end)) {
                            boolean pos = vbs.getLabel().isPositive();
                            boolean posDual = vbs2.getLabel().isPositive();
                            if ((pos && !posDual) || (!pos && posDual)) {
                                if (cbs.getLabel() != cbs2.getLabel().getInverse()) {
                                    String s = "The base "+vbs.toStringShort()+" appears equal to two distinct constants.
";
                                    log.reportEvidence(s);
                                    answer = true;
                                }
                            }
                            else {
                                if (cbs.getLabel() != cbs2.getLabel()) {
                                    String s = "The base "+vbs.toStringShort()+" appears equal to two distinct constants.
";
                                    log.reportEvidence(s);
                                    answer = true;
                                }
                            }
```

```
                        }
                    }

                }
            }
        }
        return answer;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
class GEDegeneracy_ConditionE implements IGEDegeneracyCondition {

    public boolean test(GE geq, IDegeneracyTestLog log) {
        // MK: if h_i is a variable from some coefficient equation...
        boolean answer = false;

        for (Iterator it=geq.iteratorBases(); it.hasNext();) {
            Base vbs = (Base)it.next();
            if (vbs.isConstant()) continue;

            for (Iterator it2=geq.iteratorBases(); it2.hasNext();) {
                Base cbs = (Base)it2.next();
                if ( ! cbs.isConstant()) continue;

                Boundary vbd = vbs.getBegin();
                Boundary cbd = cbs.getBegin();

                if (vbd == cbd) {
                    for (Iterator itA=geq.iteratorBases(); itA.hasNext();) {
                        Base bsA = (Base)itA.next();
                        if (bsA.isConstant()) continue;
                        Constraint consA=bsA.getConstraint();
                        for (Iterator iterA=consA.iteratorBoundary(); iterA.hasNext();) {
                            Boundary bA = (Boundary)iterA.next();
                            if (bA.getID() == vbd.getID()) {

                                for (Iterator itB=geq.iteratorBases(); itB.hasNext();) {
                                    Base bsB = (Base)itB.next();
                                    if (bsB.isConstant()) continue;
                                    Constraint consB=bsB.getConstraint();
                                    for (Iterator iterB=consB.iteratorBoundary(); iterB.hasNext();) {
                                        Boundary bB = (Boundary)iterB.next();
                                        if (bB.getID() == 1+vbd.getID()) {
```

```
                                                Boundary bA2 = consA.getDual(bA);
                                                Boundary bB2 = consB.getDual(bB);
                                                int diff = Math.abs(bA2.getID() - bB2.getID());
                                                if (diff != 1) {
                                                    String s = "The base "+vbs.toStringShort()+" must be of length 1, yet boun
dary connections in "+bsA.toStringShort()+" and "+bsB.toStringShort()+" which are co-located with its endpoints, violate t
his.  ";

                                                    log.reportEvidence(s);
                                                    answer = true;
                                                }
                                            }
                                        }
                                    }

                                }
                            }
                        }
                    }
                }
        }

        return answer;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import equation.QuadraticSystem;
import cancellation.Diagram;
import cancellation.LabeledPath;
import cancellation.Path;
import cancellation.Edge;
import letter.LetterFactory;
import letter.Letter;
import letter.Constant;
import letter.Variable;
import java.util.Iterator;
import java.util.HashMap;
import java.util.HashSet;

/**
 *
 * @author grouptheory
 */
public class GEFactory {

    private static GEFactory _instance;

    private GEFactory() {
    }

    public static GEFactory instance() {
        if (_instance == null) {
            _instance = new GEFactory();
        }
        return _instance;
    }

    public GE newGE(Diagram d, QuadraticSystem qs) {

        GE ge = new GE();
        Boundary last=ge.getFirstBoundary();

        int counter=0;
        HashMap lp2begin = new HashMap();
        HashMap lp2end = new HashMap();
```

```
        for (Iterator it = d.iteratorLabeledPaths(); it.hasNext();) {

            LabeledPath lp = (LabeledPath)it.next();

            int beginIndex=counter;
            lp2begin.put(lp,new Integer(beginIndex));

            Path p = lp.getPath();
            int len = p.length();
            for (int i=0; i<len;i++) {
                ge.appendNewBoundary();
            }

            counter +=len;

            int endIndex=beginIndex+len;
            lp2end.put(lp,new Integer(endIndex));
        }

        HashSet vars = new HashSet();

        for (Iterator it = d.iteratorLabeledPaths(); it.hasNext();) {
            LabeledPath lp = (LabeledPath)it.next();
            Letter let = lp.getLabel();
            Path p = lp.getPath();
            int len = p.length();

            int beginIndex=((Integer)lp2begin.get(lp)).intValue();
            int endIndex=((Integer)lp2end.get(lp)).intValue();
            Boundary begin = ge.getNthBoundary(beginIndex);
            Boundary end = ge.getNthBoundary(endIndex);

            if (let.isConstant()) {
                if (endIndex-beginIndex != 1) {
                    throw new RuntimeException("GEFactory.newGE: diagram contains constant base w/ length != 1");
                }
                ge.addNewConstantBase(begin, (Constant)let);
            }
            else {
                vars.add(let);

                if (endIndex-beginIndex <= 0) {
                    throw new RuntimeException("GEFactory.newGE: diagram contains variable base w/ length <= 0");
                }

                LabeledPath lp2 = d.getDual(lp);
```

```
                Letter let2;
                boolean swap = false;
                boolean forced = false;

                if (lp2 == null) {
                    HashMap lut = qs.getEquivalences();
                    Letter equiv = (Letter)lut.get(let);
                    if (equiv==null) {
                        equiv = (Letter)lut.get(let.getInverse());
                        equiv = equiv.getInverse();
                    }

                    if (equiv==null) {
                        throw new RuntimeException("GEFactory.newGE: diagram contains variable of degree 1");
                    }
                    else {
                        // System.out.println("letter "+let+" == "+equiv);

                        Variable v = (Variable)equiv;
                        lp2 = d.getVariablePath(v);
                        let2 = lp2.getLabel();

                        if (lp2 == null) {
                            throw new RuntimeException("GEFactory.newGE: unable to make a variable base");
                        }

                        if (let2 == equiv.getInverse()) {
                            swap = true;
                        }
                        forced = true;
                    }
                }
                else {
                    let2 = lp2.getLabel();

                    if (let2 == let.getInverse()) {
                        swap = true;
                    }
                    forced = false;
                }

                int beginIndex2=((Integer)lp2begin.get(lp2)).intValue();
                int endIndex2=((Integer)lp2end.get(lp2)).intValue();
                Boundary begin2 = ge.getNthBoundary(beginIndex2);
                Boundary end2 = ge.getNthBoundary(endIndex2);

                if (endIndex2-beginIndex2 <= 0) {
```

```
                throw new RuntimeException("GEFactory.newGE: diagram contains variable base w/ dual of length <= 0");
            }

            if (swap) {
                Boundary swapBdy = begin2;
                begin2 = end2;
                end2 = swapBdy;
            }

            if (forced || (lp.hashCode() < lp2.hashCode())) {
                ge.addNewVariableBase(begin, end, (Variable)let, begin2, end2);
            }
        }
    }


    for (Iterator it = d.iteratorEdges(); it.hasNext();) {
        Edge e = (Edge)it.next();
        LabeledPath[] lpArray = d.getPaths(e);

        // System.out.println("Considering edge "+e);

        LabeledPath lp1 = lpArray[0];
        LabeledPath lp2 = lpArray[1];

        // System.out.println("p1: "+lp1);
        // System.out.println("p2 "+lp2);

        int offset1 = lp1.getEdgeIndex(e);
        int offset2 = lp2.getEdgeIndex(e);
        offset1 = lp1.length() - offset1 - 1;
        offset2 = lp2.length() - offset2 - 1;

        // System.out.println("offset1: "+offset1);
        // System.out.println("offset2 "+offset2);

        int beginIndex=offset1+((Integer)lp2begin.get(lp1)).intValue();
        int endIndex=beginIndex+1;

        // System.out.println("beginIndex: "+beginIndex);

        Boundary begin = ge.getNthBoundary(beginIndex);
        Boundary end = ge.getNthBoundary(endIndex);

        int beginIndex2=offset2+((Integer)lp2begin.get(lp2)).intValue();
        int endIndex2=beginIndex2+1;
```

```
            // System.out.println("beginIndex2: "+beginIndex2);

            Boundary begin2 = ge.getNthBoundary(beginIndex2);
            Boundary end2 = ge.getNthBoundary(endIndex2);

            Variable mu = LetterFactory.instance().newUnusedVariable(vars, 100, Boolean.TRUE);
            ge.addNewVariableBase(begin, end, (Variable)mu, end2, begin2);
            vars.add(mu);
        }

        return ge;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

/**
 *
 * @author grouptheory
 */
public class GETask_BoundaryInsertion implements IGETask {
    private Boundary _bd;
    private boolean _after;
    private Boundary _newbd;
    private GE _geq;
    private String _msg;

    public GETask_BoundaryInsertion(Boundary bd, boolean after, GE geq) {
        if (bd==null) {
            throw new RuntimeException("GETask_BoundaryInsertion.ctor: bd == null");
        }

        _bd = bd;
        _after = after;
        _geq = geq;
        _msg = null;
    }

    public void execute() {
        if (_after) {
            _newbd = _geq.insertNewBoundaryAfter(_bd);
        }
        else {
            _newbd = _geq.insertNewBoundaryBefore(_bd);
        }
        _msg = log();
        // System.out.println("executing: "+_msg);
    }

    public Boundary getNewBoundary() {
        return _newbd;
    }

    public String toString() {
        return _msg;
    }
```

```
    private String log() {
        String s="";
        s += "Added (new) boundary "+_newbd+".";
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

/**
 *
 * @author grouptheory
 */
public class GETask_CollapseBase implements IGETask {

    private Base _bs;
    private GE _geq;
    private String _msg;

    public GETask_CollapseBase(Base bs, GE geq) {
        if (bs==null) {
            throw new RuntimeException("GETask_CollapseBase.ctor: bs == null");
        }

        if (bs.getLabel().isConstant()) {
            throw new RuntimeException("GETask_CollapseBase.ctor: bs == constant");
        }

        Base dual = bs.getDual();

        if (!dual.isEmpty() &&
           ((bs.getBegin() != dual.getBegin()) ||
            (bs.getEnd() != dual.getEnd()) ||
            (bs.getLabel() != dual.getLabel()))) {
            throw new RuntimeException("GETask_CollapseBase.ctor: bs not collapsible");
        }

        _bs = bs;
        _geq = geq;
        _msg = log();
    }

    public void execute() {
        // System.out.println("executing: "+_msg);
        _geq.collapseBase(_bs);
    }

    public String toString() {
        return _msg;
```

```
    }

    private String log() {
        String s="";
        s += "Collapsing (new) base "+_bs.toStringShort()+" to the empty base ("+_bs.getEnd()+","+_bs.getEnd()+").\n";
        return s;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

/**
 *
 * @author grouptheory
 */
public class GETask_ConstraintAddition implements IGETask {
    private Base _bs;
    private Boundary _bd;
    private Boundary _bd_dual;
    private GE _geq;
    private String _msg;

    public GETask_ConstraintAddition(Base bs, Boundary bd, Boundary bd_dual, GE geq) {
        if (bs==null) {
            throw new RuntimeException("GETask_ConstraintAddition.ctor: bs == null");
        }
        if (bd==null) {
            throw new RuntimeException("GETask_ConstraintAddition.ctor: bd == null");
        }
        if (bd_dual==null) {
            throw new RuntimeException("GETask_ConstraintAddition.ctor: bd_dual == null");
        }

        _bs = bs;
        _bd = bd;
        _bd_dual = bd_dual;
        _geq = geq;
        _msg = null;
    }

    public void execute() {
        // System.out.println("executing: "+log());
        _geq.addNewConstraint(_bs, _bd, _bd_dual);
        _msg = log();
    }

    public String toString() {
        return _msg;
    }

    private String log() {
```

```
        String s="";
        s += "Added constraint between boundary "+_bd+" in (new) base "+_bs.toStringShort()+" and boundary "+_bd_dual+" in
 its dual.";
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

/**
 *
 * @author grouptheory
 */
public class GETask_ConstraintDeletion implements IGETask {
    private Base _bs;
    private Boundary _bd;
    private Boundary _bd_dual;
    private GE _geq;
    private String _msg;

    public GETask_ConstraintDeletion(Base bs, Boundary bd, Boundary bd_dual, GE geq) {
        if (bs==null) {
            throw new RuntimeException("GETask_ConstraintDeletion.ctor: bs == null");
        }
        if (bd==null) {
            throw new RuntimeException("GETask_ConstraintDeletion.ctor: bd == null");
        }
        if (bd_dual==null) {
            throw new RuntimeException("GETask_ConstraintDeletion.ctor: bd_dual == null");
        }

        _bs = bs;
        _bd = bd;
        _bd_dual = bd_dual;
        _geq = geq;
        _msg = log();
    }

    public void execute() {
        // System.out.println("executing: "+_msg);
        _geq.removeConstraint(_bs, _bd, _bd_dual);
    }

    public String toString() {
        return _msg;
    }

    private String log() {
        String s="";
```

```
        s += "Deleted constraint between boundary "+_bd+" in (old) base "+_bs.toStringShort()+" and boundary "+_bd_dual+"
in its dual.";
        return s;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

/**
 *
 * @author grouptheory
 */
public class GETask_DeleteBase implements IGETask {

    private Base _bs;
    private GE _geq;
    private String _msg;

    public GETask_DeleteBase(Base bs, GE geq) {
        if (bs==null) {
            throw new RuntimeException("GETask_DeleteBase.ctor: bs == null");
        }
        _bs = bs;
        _geq = geq;
        _msg = log();
    }

    public void execute() {
        // System.out.println("executing: "+_msg);
        _geq.removeBase(_bs);
    }

    public String toString() {
        return _msg;
    }

    private String log() {
        String s="";
        s += "Deleting (new) base "+_bs.toStringShort()+" because it begins to the left of the critical boundary.\n";
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

/**
 *
 * @author grouptheory
 */
public class GETask_DeleteBoundary implements IGETask {

    private Boundary _bd;
    private GE _geq;
    private String _msg;

    public GETask_DeleteBoundary(Boundary bd, GE geq) {
        if (bd==null) {
            throw new RuntimeException("GETask_DeleteBoundary.ctor: bd == null");
        }
        _bd = bd;
        _geq = geq;
        _msg = log();
    }

    public void execute() {
        // System.out.println("executing: "+_msg);
        _geq.removeBoundary(_bd);
    }

    public String toString() {
        return _msg;
    }

    private String log() {
        String s="";
        s += "Deleting (new) boundary "+_bd+" because it is not used inside any base.  This will cause renumbering of high
er numbered boundaries.\n";
        return s;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

/**
 *
 * @author grouptheory
 */
public class GETask_MoveBase implements IGETask {
    private Base _bs;
    private Boundary _begin;
    private Boundary _end;
    private GE _geq;
    private String _msg;

    public GETask_MoveBase(Base bs, Boundary begin, Boundary end, GE geq) {
        if (begin==null) {
            throw new RuntimeException("GETask_MoveBase.ctor: begin == null");
        }
        if (end==null) {
            throw new RuntimeException("GETask_MoveBase.ctor: end == null");
        }
        if (bs==null) {
            throw new RuntimeException("GETask_MoveBase.ctor: bs == null");
        }

        _bs = bs;
        _begin = begin;
        _end = end;
        _geq = geq;
        _msg = log1();
    }

    public void execute() {
        // System.out.println("executing: "+_msg);
        _bs.move(_begin, _end);
        _msg += log2();
    }

    public String toString() {
        return _msg;
    }

    private String log1() {
```

```
        String s="";
        s += "Moved (old) base "+_bs.toStringShort()+"";
        return s;
    }

    private String log2() {
        String s="";
        s += " to (new) boundaries "+_begin+" - "+_end+".";
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

/**
 *
 * @author grouptheory
 */
public interface IDegeneracyTestLog {
    void reportEvidence(String s);
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

/**
 *
 * @author grouptheory
 */
interface IGEDegeneracyCondition {
    boolean test(GE geq, IDegeneracyTestLog log);
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

/**
 *
 * @author grouptheory
 */
public interface IGETask {
    void execute();
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import java.util.TreeMap;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;
import letter.Letter;
import letter.Constant;
import letter.Variable;

/**
 *
 * @author grouptheory
 */
public class Latex {

    private static Latex _instance;

    private Latex() {
    }

    public static Latex instance() {
        if (_instance == null) {
            _instance = new Latex();
        }
        return _instance;
    }

    public String renderGEasText(GE ge) {
        String s = "";
        s+="\\begin{verbatim}\n";
        s+=ge.toString()+"\n";
        s+="\\end{verbatim}\n";
        return s;
    }

    public String renderGEasGraphics(GE ge) {
        int numBD = ge.getNumberOfBoundaries();
        int numBS = ge.getNumberOfBases();

        int START_ITEM_BASE = 1000;
        int START_QUADRATIFY_BASE = 5000;
```

```
        double WIDTH = 7.0;
        double HEIGHT = 6.0;

        String s = "";
        s+="\\begin{center}\n";
        s+="\\begin{pspicture}(-0.5,-0.5)(7.5,6.5)\n";

        if (numBD==1) {

        }
        else {
            double boundaryspace = BaseLayoutDecoratorFactory.applyToAllBases(ge, WIDTH, HEIGHT);

            double x = 0.0;
            for (int i=0; i<numBD; i++) {
                s+="\\psline[linecolor=black]{-}("+x+",0.0)("+x+",6.0)";
                s+="\\rput{0}("+x+",0.0){$"+i+"$}\n";
                s+="\\rput{0}("+x+","+HEIGHT+"){$"+i+"$}\n";
                x += boundaryspace;
            }

            BaseComparator bcomp = new BaseComparator();

            for (Iterator it=ge.iteratorBases(); it.hasNext();) {
                Base bs=(Base)it.next();
                BaseLayoutDecorator bld = (BaseLayoutDecorator)bs.lookupDecorator(BaseLayoutDecorator.NAME);
                double left = bld.getX1();
                double right = bld.getX2();
                double mid = (left+right)/2.0;
                String arrowhead = "{[->}";
                Letter let = bs.getLabel();
                if (!let.isPositive()) {
                    arrowhead = "{<-]}";
                }

                double y = bld.getY();
                //System.out.println("assigned "+bs+" y="+y);

                if (let.isConstant()) {
                    String colorStr="";
                    if (let.isConstant()) {
                        if (let.modulus(3)==0) {
                            colorStr="yellow";
                        }
                        else if (let.modulus(3)==1) {
                            colorStr="blue";
```

```
                }
                else {
                    colorStr="green";
                }
            }
            else {
                colorStr="red";
            }
            s+="\\psline[linecolor="+colorStr+"]"+arrowhead+"("+left+","+y+")("+right+","+y+")";

            if (!let.isPositive()) {
                let = let.getInverse();
            }
            y -= 0.2;
            s+="\\rput{0}("+mid+","+y+"){$"+letter.Latex.instance().render(let)+"$}\n";
        }
        else {
            String colorStr="";
            colorStr="red";
            s+="\\psline[linecolor="+colorStr+"]"+arrowhead+"("+left+","+y+")("+right+","+y+")";

            if (!let.isPositive()) {
                let = let.getInverse();
            }
            y -= 0.2;
            s+="\\rput{0}("+mid+","+y+"){$"+letter.Latex.instance().render(let)+"$}\n";

            Base bs2 = bs.getDual();
            if (bcomp.compare(bs, bs2) > 0) {

                BaseLayoutDecorator bld2 = (BaseLayoutDecorator)bs2.lookupDecorator(BaseLayoutDecorator.NAME);
                double left2 = bld2.getX1();
                double right2 = bld2.getX2();
                double y2 = bld2.getY();

                int conidx=0;
                Constraint con = bs.getConstraint();

                for (Iterator it2=con.iteratorBoundary(); it2.hasNext();) {
                    Boundary bd1 = (Boundary)it2.next();
                    Boundary bd2 = con.getDual(bd1);
                    double x1 = left+(right-left)*(double)(bd1.getID()-bs.getBegin().getID())/
                                                  (double)(bs.getEnd().getID()-bs.getBegin().getID());
                    double x2 = left2+(right2-left2)*(double)(bd2.getID()-bs2.getBegin().getID())/
                                                     (double)(bs2.getEnd().getID()-bs2.getBegin().getID());
                    String fillcolor = "";
                    if (conidx%2 == 0) {
```

```
                                fillcolor = "black";
                            }
                            else {
                                fillcolor = "white";
                            }

                            s+="\\pscircle[linecolor=red,fillcolor="+fillcolor+",fillstyle=solid]("+x1+","+(y+0.2)+"){0.07
5}\n";
                            s+="\\pscircle[linecolor=red,fillcolor="+fillcolor+",fillstyle=solid]("+x2+","+y2+"){0.075}\n"
;
                            conidx++;
                        }
                    }
                }
            }
        }
        s+="\\end{pspicture}\n";
        s+="\\end{center}\n";

        return s;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import letter.Letter;
import letter.Constant;
import letter.Variable;
import letter.LetterFactory;
import equation.GroupEquation;
import equation.QuadraticSystem;
import cancellation.ICancellationDiagramAnalysis;
import cancellation.CancellationDiagramFactory;
import cancellation.Diagram;
import cancellation.DiagramTreeNode;
import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
public class Main {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

        Constant a = LetterFactory.instance().getConstant(1, true);
        Constant b = LetterFactory.instance().getConstant(2, true);
        Constant A = (Constant)a.getInverse();
        Constant B = (Constant)b.getInverse();

        Variable z1 = LetterFactory.instance().getVariable(1, true);
        Variable z2 = LetterFactory.instance().getVariable(2, true);

        GE ge1 = new GE();
        Boundary b0 = ge1.getFirstBoundary();
        Boundary b1 = ge1.appendNewBoundary();
        Boundary b2 = ge1.appendNewBoundary();
        Boundary b3 = ge1.appendNewBoundary();
        Base c1 = ge1.addNewConstantBase(b0, a);
        Base c2 = ge1.addNewConstantBase(b1, b);
        Base c3 = ge1.addNewConstantBase(b2, A);
        Base c4 = ge1.addNewConstantBase(b3, B);
```

```
        Base v1 = ge1.addNewVariableBase(b0, b2, z1, b1, b3);
        Base v2 = ge1.addNewVariableBase(b0, b1, z2, b3, b2);

        ge1.addNewConstraint(v1, b1, b2);

        System.out.println("GE#1:\n");
        System.out.println(""+ge1+"\n");

        GE ge2 = ge1.duplicate();

        System.out.println("GE#1:\n");
        System.out.println(""+ge1+"\n");

        System.out.println("GE#2:\n");
        System.out.println(""+ge2+"\n");




        System.out.println("\nTEST Real-world EQUATION TEST:\n");

        GroupEquation prob = new GroupEquation("z1+.c1+.z1+.c2+.z1-.");
        System.out.println("Original Equation: "+prob+" = 1\n");

        ICancellationDiagramAnalysis analysis =
                CancellationDiagramFactory.instance().newDiagramTree(prob);

        System.out.println("Analysis: \n\n");
        QuadraticSystem qs = analysis.getQuadraticSystem();
        System.out.println(qs);

        Iterator it = analysis.iteratorDiagramTreeNodes();
        DiagramTreeNode dtn = (DiagramTreeNode)it.next();
        Diagram d = dtn.getDiagram();

        System.out.println("Diagram:\n");
        System.out.println(""+d+"\n\n");

        GEFactory gef = GEFactory.instance();
        GE ge3 = gef.newGE(d, qs);

        System.out.println("GE#3:\n");
        System.out.println(""+ge3+"\n");

        GE ge4 = ge3.duplicate();

        System.out.println("GE#4:\n");
```

```
        System.out.println(""+ge4+"\n");
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import equation.GroupEquation;

/**
 *
 * @author grouptheory
 */
public class Main2 {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

        GroupEquation problem = new GroupEquation("z1+.c1+.z2+.c2+.z1-.c3+.z2-.");

        TopLevelGEIterator iter = TopLevelGEIteratorFactory.instance().newTopLevelGEIterator(problem);

        int i=0;
        for (;iter.hasNext();) {
            GE geq = (GE)iter.next();

            System.out.println("GE#"+i+":\n");
            System.out.println(""+geq+"\n");
            i++;
        }

        System.out.println("Total number of GEs generated: "+i+"\n");
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import equation.GroupEquation;
import equation.QuadraticSystem;
import equation.QuadraticSystemFactory;
import cancellation.Diagram;
import cancellation.DiagramDegeneracyTester;
import cancellation.ComposableDiagramIterator;
import cancellation.DiagramTreeNode;
import utility.CompositeIterator;

/**
 *
 * @author grouptheory
 */
public class TopLevelGEIterator {

    private GroupEquation _eq;
    private QuadraticSystem _qs;
    private GroupEquation _problemQuadratic;
    private Diagram _rootDiagram;
    private ComposableDiagramIterator _cdi;
    private CompositeIterator _compiter;

    private GE _nextGE;
    private boolean _finished;

    TopLevelGEIterator(GroupEquation eq) {
        _eq = new GroupEquation(eq);
        _qs = QuadraticSystemFactory.instance().newQuadraticSystem(_eq);
        _problemQuadratic = _qs.getEquation();
        _rootDiagram = new Diagram();
        _cdi = new ComposableDiagramIterator(null, _rootDiagram, _problemQuadratic, 0);
        _compiter = new CompositeIterator(_cdi, false);

        _nextGE = nextGE();
    }

    private GE nextGE() {
        boolean finished = false;
        GE geq = null;
```

```
        GEDegeneracyTester tester;
        do {
            tester = new GEDegeneracyTester(geq);
            DiagramTreeNode nextDTN = nextDiagramTreeNode();
            if (nextDTN != null) {
                Diagram nextDiag = nextDTN.getDiagram();
                GEFactory gef = GEFactory.instance();
                geq = gef.newGE(nextDiag, _qs);
            }
            else {
                finished=true;
                break;
            }
        }
        while (tester.isDegenerate());

        if (finished) {
            return null;
        }
        else {
            return geq;
        }
    }

    private DiagramTreeNode nextDiagramTreeNode() {
        DiagramTreeNode answer = null;
        while (_compiter.hasNext()) {
            CompositeIterator.State state = (CompositeIterator.State)_compiter.next();
            DiagramTreeNode dtn = (DiagramTreeNode)state.getLeafIteratorState();
            Diagram nextDiag = dtn.getDiagram();

            if (!DiagramDegeneracyTester.isDegenerate(nextDiag) && dtn.getLeaf()) {
                answer = dtn;
                break;
            }
        }
        return answer;
    }

    public boolean hasNext() {
        return (_nextGE!=null);
    }

    public Object next() {
        GE nextGE = _nextGE;
        _nextGE = nextGE();
        return nextGE;
```

```
    }

    public void remove() {
        throw new RuntimeException("TopLevelGEIterator.remove: not implemented");
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ge;

import equation.GroupEquation;

/**
 *
 * @author grouptheory
 */
public class TopLevelGEIteratorFactory {

    private static TopLevelGEIteratorFactory _instance;

    private TopLevelGEIteratorFactory() {
    }

    public static TopLevelGEIteratorFactory instance() {
        if (_instance == null) {
            _instance = new TopLevelGEIteratorFactory();
        }
        return _instance;
    }

    public TopLevelGEIterator newTopLevelGEIterator(GroupEquation eq) {
        return new TopLevelGEIterator(eq);
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package hom;

import equation.GroupEquation;

/**
 *
 * @author grouptheory
 */
public class Hom {
    public GroupEquation apply(GroupEquation eq) {
        return eq;
    }

    static Hom compose(Hom hfirst, Hom hsecond) {
        return null;
    }
}
```

```
package jigglecore;

/* Abstract base class for all JIGGLE objects that have rectangular
representations.  Known subclasses: Vertex, EdgeLabel, QuadTree. */

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public abstract class Cell extends JiggleObject {

	private int dimensions = 2; /* default is a 2-D cell */
	private double weight; /* weight of cell */
	private double coords []; /* coordinates of center of cell */
	private double min [], max []; /* bounding box of cell */
	private double size []; /* dimensions of cell */

	protected Cell () {setDimensions (2); weight = 0;}

	double getWeight () {return weight;}
	void setWeight (double w) {weight = w;}

	public int getDimensions () {return dimensions;}
	public void setDimensions (int d) {
		dimensions = d; coords = new double [d]; size = new double [d];
		min = new double [d]; max = new double [d];
	}

	public double [] getCoords () {return coords;}
	public void setCoords (double [] c) {
		for (int i = 0; i < dimensions; i++) coords [i] = c [i];
	}

	double [] getMin () {return min;}
	void setMin (double [] c) {
		for (int i = 0; i < dimensions; i++) min [i] = c [i];
		recomputeSize ();
	}

	double [] getMax () {return max;}
	void setMax (double [] c) {
		for (int i = 0; i < dimensions; i++) max [i] = c [i];
		recomputeSize ();
	}

	protected void recomputeSize () {
		for (int i = 0; i < dimensions; i++) size [i] = max [i] - min [i];
```

```
	}

	public double [] getSize () {return size;}
	void setSize (double [] c) {
		for (int i = 0; i < dimensions; i++) size [i] = c [i];
		recomputeBoundaries ();
	}

	void recomputeBoundaries () {
		for (int i = 0; i < dimensions; i++) {
			min [i] = coords [i] - size [i] / 2;
			max [i] = coords [i] + size [i] / 2;
		}
	}

	void translate (double [] vector) {translate (1, vector);}
	void translate (double scalar, double [] vector) {
		for (int i = 0; i < dimensions; i++) {
			double translation = scalar * vector [i];
			coords [i] += translation;
			min [i] += translation;
			max [i] += translation;
		}
	}

	static double getDistanceSquared (Cell c1, Cell c2) {
		double sum = 0; int d = c1.getDimensions ();
		for (int i = 0; i < d; i++)
			sum += square (c1.coords [i] - c2.coords [i]);
		return sum;
	}

	static double getDistanceSquared (Cell cell, double [] point) {
		double sum = 0; int d = cell.getDimensions ();
		for (int i = 0; i < d; i++)
			sum += square (cell.coords [i] - point [i]);
		return sum;
	}

	static double getDistance (Cell c1, Cell c2) {
		return Math.sqrt (getDistanceSquared (c1, c2));
	}

	static double getDistance (Cell cell, double [] point) {
		return Math.sqrt (getDistanceSquared (cell, point));
	}
```

```
	static double sumOfRadii (Cell c1, Cell c2) {
		int d = c1.getDimensions ();
		double coords1 [] = c1.getCoords (), coords2 [] = c2.getCoords ();
		double seg [] = new double [d];
		for (int i = 0; i < d; i++) seg [i] = coords2 [i] - coords1 [i];
		return radius (d, c1.getSize (), seg) + radius (d, c2.getSize (), seg);
	}

	static double radius (Cell cell, double [] point) {
		int d = cell.getDimensions ();
		double coords [] = cell.getCoords ();
		double seg [] = new double [d];
		for (int i = 0; i < d; i++) seg [i] = point [i] - coords [i];
		return radius (d, cell.getSize (), seg);
	}

	private static double radius (int d, double [] cellSize, double [] segment) {
		double sum = 0;
		for (int i = 0; i < d; i++) sum += cellSize [i];
		if (sum == 0) return 0;
		double t = Double.MAX_VALUE;
		for (int i = 0; i < d; i++) {
			t = Math.min (t, Math.abs (cellSize [i] / segment [i]));
		}
		double lengthSquared = 0;
		for (int i = 0; i < d; i++) lengthSquared += square (t * segment [i]);
		return Math.sqrt (lengthSquared) / 2;
	}
}
```

```java
package jigglecore;

/* Class for conjugate gradient method. */

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class ConjugateGradients extends FirstOrderOptimizationProcedure {

	private double magnitudeOfPreviousGradientSquared;
	private double previousDescentDirection [] [] = null;
	private double restartThreshold = 0;

	public ConjugateGradients (Graph g, ForceModel fm, double acc) {
		super (g, fm, acc); restartThreshold = 0;
	}

	public ConjugateGradients (Graph g, ForceModel fm, double acc, double rt) {
		super (g, fm, acc); restartThreshold = rt;
	}

	public void reset () {negativeGradient = null; descentDirection = null;}

	protected void computeDescentDirection () {
		int n = graph.numberOfVertices, d = graph.getDimensions ();
		double magnitudeOfCurrentGradientSquared = 0;
		if ((descentDirection == null) || (descentDirection.length != n)) {
			descentDirection = new double [n] [d];
			previousDescentDirection = new double [n] [d];
			for (int i = 0; i < n; i++) {
				for (int j = 0; j < d; j++) {
					double temp = negativeGradient [i] [j];
					descentDirection [i] [j] = temp;
					magnitudeOfCurrentGradientSquared += square (temp);
				}
			}
		}
		else {
			for (int i = 0; i < n; i++) {
				for (int j = 0; j < d; j++) {
					double temp = negativeGradient [i] [j];
					magnitudeOfCurrentGradientSquared += square (temp);
				}
			}
			if (magnitudeOfCurrentGradientSquared < 0.000001) {
				for (int i = 0; i < n; i++) {
```

```
                                        for (int j = 0; j < d; j++) {
                                                previousDescentDirection [i] [j] = 0;
                                                descentDirection [i] [j] = 0;
                                        }
                                }
                                return;
                        }
                        double w = magnitudeOfCurrentGradientSquared / magnitudeOfPreviousGradientSquared;
                        double dotProduct = 0, magnitudeOfDescentDirectionSquared = 0, m;
                        for (int i = 0; i < n; i++) {
                                for (int j = 0; j < d; j++) {
                                        descentDirection [i] [j] = negativeGradient [i] [j] +
                                                                   w * previousDescentDirection [i] [j];
                                        dotProduct += descentDirection [i] [j] * negativeGradient [i] [j];
                                        magnitudeOfDescentDirectionSquared += square (descentDirection [i] [j]);
                                }
                        }
                        m = magnitudeOfCurrentGradientSquared * magnitudeOfDescentDirectionSquared;
                        if (dotProduct / Math.sqrt (m) < restartThreshold) {
                                descentDirection = null; computeDescentDirection (); return;
                        }
                }
                magnitudeOfPreviousGradientSquared = magnitudeOfCurrentGradientSquared;
                for (int i = 0; i < n; i++) {
                        for (int j = 0; j < d; j++) {
                                previousDescentDirection [i] [j] = descentDirection [i] [j];
                        }
                }
        }
}
```

```
package jigglecore;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public abstract class Constraint extends JiggleObject {

        protected Graph graph;

        protected Constraint (Graph g) {graph = g;}

        abstract void apply (double [][] penalty);
}
```

```java
package jigglecore;

/* Methods for manipulating dynamic arrays of JiggleObjects. */

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public abstract class DynamicArray {

        public static Vertex [] add (Vertex [] arr, int size, Vertex elem) {
                if (size == arr.length) {
                        Vertex newArr [] = new Vertex [2 * size];
                        for (int i = 0; i < size; i++) newArr [i] = arr [i];
                        arr = newArr;
                }
                arr [size] = elem; return arr;
        }

        public static Edge [] add (Edge [] arr, int size, Edge elem) {
                if (size == arr.length) {
                        Edge newArr [] = new Edge [2 * size];
                        for (int i = 0; i < size; i++) newArr [i] = arr [i];
                        arr = newArr;
                }
                arr [size] = elem; return arr;
        }

        public static void remove (JiggleObject [] arr, int size, JiggleObject elem)
                        throws NotFoundException {
                for (int i = 0; i < size; i++)
                        if (arr [i] == elem) {arr [i] = arr [size - 1]; return;}
                throw new NotFoundException ();
        }
}
```

```
package jigglecore;

/* Class for edges of a graph.  NOTE: the only mutable characteristics
of an edge are its label, directedness, and preferred length. */

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class Edge extends JiggleObject {

	private Vertex from, to; /* endpoints of the edge */
	private EdgeLabel label = null; /* label of edge */
	private boolean directed = false; /* is the edge directed? */
	private double preferredLength = 0; /* preferred length of edge */

	Edge (Graph g, Vertex f, Vertex t) {from = f; to = t; setContext (g);}

	Edge (Graph g, Vertex f, Vertex t, boolean dir) {
		from = f; to = t; setContext (g); directed = dir;
	}

	public Vertex getFrom () {return from;}
	public Vertex getTo () {return to;}

	EdgeLabel getLabel () {return label;}
	void setLabel (EdgeLabel lbl) {label = lbl;}

	boolean getDirected () {return directed;}
	void setDirected (boolean d) {directed = d;}

	double getPreferredLength () {return preferredLength;}
	public void setPreferredLength (double len) {preferredLength = len;}

	double getLengthSquared () {return Vertex.getDistanceSquared (from, to);}
	double getLength () {return Vertex.getDistance (from, to);}

	public String toString () {
		return "(Edge: " + from + ", " + to + ", " +
			(directed ? "directed" : "undirected") + ")";
	}
}
```

```
package jigglecore;

/* Class for edge labels. */

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class EdgeLabel extends Cell {

        String name;

        EdgeLabel (Edge e, String str) {setContext (e); name = str;}

        String getName () {return name;}
        void setName (String str) {name = str;}

        public String toString () {return "(EdgeLabel: " + name + ")";}
}
```

```
package jigglecore;

/* Abstract base class for first-order graph-drawing optimization procedures.
Includes concrete method for performing adaptive line search. */

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public abstract class FirstOrderOptimizationProcedure extends ForceDirectedOptimizationProcedure {

	protected double maxCos = 1;

	FirstOrderOptimizationProcedure (Graph g, ForceModel fm, double accuracy) {
		super (g, fm); maxCos = accuracy;
	}

	protected double negativeGradient [] [] = null;
	protected double descentDirection [] [] = null;
	protected double penaltyVector [] [] = null;
	protected double penaltyFactor = 0;

	public double improveGraph () {
		int n = graph.numberOfVertices, d = graph.getDimensions ();
		if ((negativeGradient == null) || (negativeGradient.length != n)) {
			negativeGradient = new double [n] [d];
			penaltyVector = new double [n] [d];
			getNegativeGradient ();
		}
		computeDescentDirection ();
		return lineSearch ();
	}

	public void reset () {negativeGradient = null; penaltyFactor = 0;}

	private void computePenaltyFactor () {
		double m1 = l2Norm (negativeGradient);
		double m2 = l2Norm (penaltyVector);
		if (m2 == 0) penaltyFactor = 0;
		else if (m1 == 0) penaltyFactor = 1;
		else {
			double cos = dotProduct (negativeGradient, penaltyVector) / (m1*m2);
			penaltyFactor = Math.max (0.00000001, (0.00000001 - cos)) * Math.max (1, (m1 / m2));
		}
	}

	private void getNegativeGradient () {
```

```
		forceModel.getNegativeGradient (negativeGradient);
		if (constrained) {
			getPenaltyVector (); computePenaltyFactor ();
			int n = graph.numberOfVertices, d = graph.getDimensions ();
			for (int i = 0; i < n; i++) {
				for (int j = 0; j < d; j++) {
					negativeGradient [i] [j] += penaltyFactor * penaltyVector [i] [j];
				}
			}
		}
	}

	private void getPenaltyVector () {
		forceModel.getPenaltyVector (penaltyVector);
	}

	protected abstract void computeDescentDirection ();

	private double stepSize = 0.1, previousStepSize = 0;

	protected double lineSearch () {
		previousStepSize = 0;
		int n = graph.numberOfVertices;
		double magDescDir = l2Norm (descentDirection);
		if (magDescDir < 0.0001) return 0;
		double magLo = l2Norm (negativeGradient);
		step (); getNegativeGradient ();
		double magHi = l2Norm (negativeGradient);
		double m = magDescDir * magHi;
		double cos = dotProduct (negativeGradient, descentDirection) / m;
		double lo = 0, hi = Double.MAX_VALUE;
		int i = 0;
		while (((cos < 0) || (cos > maxCos)) && (hi - lo > 0.00000001)) {
			if (cos < 0) {hi = stepSize; stepSize = (lo+hi)/2;}
			else {
				if (hi < Double.MAX_VALUE) {lo = stepSize; stepSize = (lo+hi)/2;}
				else {lo = stepSize; stepSize *= 2;}
			}
			step (); getNegativeGradient ();
			m = magDescDir * l2Norm (negativeGradient);
			cos = dotProduct (negativeGradient, descentDirection) / m;
		}
		return l2Norm (negativeGradient);
	}

	private void step () {
		int n = graph.numberOfVertices;
```

```
                double s = stepSize - previousStepSize;
                for (int i = 0; i < n; i++)
                        graph.vertices [i].translate (s, descentDirection [i]);
                previousStepSize = stepSize;
        }

        protected double dotProduct (double [] [] u, double [] [] v) {
                int n = graph.numberOfVertices, d = graph.getDimensions ();
                double sum = 0;
                for (int i = 0; i < n; i++) {
                        for (int j = 0; j < d; j++) {
                                sum += u [i] [j] * v [i] [j];
                        }
                }
                return sum;
        }

        protected double l2Norm (double [] [] vect) {
                return Math.sqrt (dotProduct (vect, vect));
        }

        protected double lInfinityNorm (double [] [] vect) {
                int n = graph.numberOfVertices, d = graph.getDimensions ();
                double max = 0;
                for (int i = 0; i < n; i++) {
                        for (int j = 0; j < d; j++) {
                                max = Math.max (max, Math.abs (vect [i] [j]));
                        }
                }
                return max;
        }
}
```

```
package jigglecore;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public abstract class ForceDirectedOptimizationProcedure extends JiggleObject {

        protected Graph graph;
        protected ForceModel forceModel;

        protected boolean constrained = false;
        public boolean getConstrained () {return constrained;}
        public void setConstrained (boolean c) {constrained = c;}

        ForceDirectedOptimizationProcedure (Graph g, ForceModel fm) {
                graph = g; forceModel = fm;
        }

        public abstract double improveGraph ();
}
```

```
package jigglecore;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public abstract class ForceLaw extends JiggleObject {

        abstract void apply (double [][] negativeGradient);

        protected Graph graph;

        protected ForceLaw (Graph g) {graph = g;}

        protected double cap = Double.MAX_VALUE / 1000;
        double getCap () {return cap;}
        void setCap (double c) {cap = c;}
}
```

```
package jigglecore;

import java.util.Vector;
import java.util.Enumeration;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class ForceModel {

	protected Graph graph = null;
	protected double preferredEdgeLength;

	private Vector forceLaws = new Vector ();
	private Vector constraints = new Vector ();

	public ForceModel (Graph g) {graph = g;}

	double getPreferredEdgeLength () {return preferredEdgeLength;}
	void setPreferredEdgeLength (double k) {preferredEdgeLength = k;}

	public void addForceLaw (ForceLaw fl) {forceLaws.addElement (fl);}
	public void removeForceLaw (ForceLaw fl) {forceLaws.removeElement (fl);}

	public void addConstraint (Constraint c) {constraints.addElement (c);}
	public void removeConstraint (Constraint c) {constraints.removeElement (c);}

	void getNegativeGradient (double [] [] negativeGradient) {
		int n = graph.numberOfVertices, d = graph.getDimensions ();
		for (int i = 0; i < n; i++) {
			for (int j = 0; j < d; j++) {
				negativeGradient [i] [j] = 0;
			}
			graph.vertices [i].intField = i;
		}
		for (Enumeration en = forceLaws.elements (); en.hasMoreElements ();)
			((ForceLaw) (en.nextElement ())).apply (negativeGradient);
	}

	void getPenaltyVector (double [] [] penaltyVector) {
		int n = graph.numberOfVertices, d = graph.getDimensions ();
		for (int i = 0; i < n; i++) {
			for (int j = 0; j < d; j++) {
				penaltyVector [i] [j] = 0;
			}
			graph.vertices [i].intField = i;
```

```
                }
                for (Enumeration en = constraints.elements (); en.hasMoreElements ();)
                        ((Constraint) (en.nextElement ())).apply (penaltyVector);
        }
}
```

```
package jigglecore;

// Class for graphs. */

import java.util.*;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class Graph extends Cell {

        public int numberOfVertices = 0, numberOfEdges = 0;
        public Vertex vertices [] = new Vertex [1];
        public Edge edges [] = new Edge [1];

        /* NOTE: the above are made publicly accessible for reasons of
        efficiency.  They should NOT, however, be modified except by
        insertVertex, deleteVertex, insertEdge, and deleteEdge methods
        below. */

        public Graph () {}
        public Graph (int d) {setDimensions (d);}

        public Vertex insertVertex () {
                Vertex v = new Vertex (this);
                vertices = DynamicArray.add (vertices, numberOfVertices++, v);
                return v;
        }

        public Edge insertEdge (Vertex from, Vertex to) {
                return insertEdge (from, to, false);
        }

        public Edge insertEdge (Vertex from, Vertex to, boolean dir) {
                Edge e = new Edge (this, from, to, dir);
                from.insertNeighbor (e); to.insertNeighbor (e);
                edges = DynamicArray.add (edges, numberOfEdges++, e);
                return e;
        }

        public void deleteVertex (Vertex v) {
                try {
                        for (int i = 0; i < v.inDegree; i++) {
                                Edge e = v.undirectedEdges [i];
                                v.undirectedNeighbors [i].deleteNeighbor (e);
                                DynamicArray.remove (edges, numberOfEdges--, e);
```

```
                        }
                        for (int i = 0; i < v.inDegree; i++) {
                                Edge e = v.inEdges [i];
                                v.inNeighbors [i].deleteNeighbor (e);
                                DynamicArray.remove (edges, numberOfEdges--, e);
                        }
                        for (int i = 0; i < v.outDegree; i++) {
                                Edge e = v.outEdges [i];
                                v.outNeighbors [i].deleteNeighbor (e);
                                DynamicArray.remove (edges, numberOfEdges--, e);
                        }
                        DynamicArray.remove (vertices, numberOfVertices--, v);
                } catch (NotFoundException exc) {throw new Error (v + " not found");}
        }

        public void deleteEdge (Edge e) {
                try {
                        e.getFrom ().deleteNeighbor (e); e.getTo ().deleteNeighbor (e);
                        DynamicArray.remove (edges, numberOfEdges--, e);
                } catch (NotFoundException exc) {throw new Error (e + " not found");}
        }

        void recomputeBoundaries () {
                int d = getDimensions ();
                double lo [] = getMin (), hi [] = getMax ();
                for (int i = 0; i < d; i++) {
                        lo [i] = Double.MAX_VALUE; hi [i] = -Double.MAX_VALUE;
                }
                for (int i = 0; i < numberOfVertices; i++) {
                        Vertex v = vertices [i]; double c [] = v.getCoords ();
                                for (int j = 0; j < d; j++) {
                                        lo [j] = Math.min (lo [j], c [j]);
                                        hi [j] = Math.max (hi [j], c [j]);
                                }
                }
                recomputeSize ();
        }

        // The isConnected method tests whether a graph is connected.
        // An empty graph is considered to be not connected.

        boolean isConnected () {
                if (numberOfVertices == 0) return false;
                for (int i = 0; i < numberOfVertices; i++)
                        vertices [i].booleanField = false;
                numberOfMarkedVertices = 0;
                dft (vertices [0]);
```

```
		return (numberOfMarkedVertices == numberOfVertices);
	}
	private int numberOfMarkedVertices = 0;
	private void dft (Vertex v) {
		v.booleanField = true; ++numberOfMarkedVertices;
		for (int i = 0; i < v.undirectedDegree; i++) {
			Vertex neighbor = v.undirectedNeighbors [i];
			if (! neighbor.booleanField) dft (neighbor);
		}
		for (int i = 0; i < v.undirectedDegree; i++) {
			Vertex neighbor = v.inNeighbors [i];
			if (! neighbor.booleanField) dft (neighbor);
		}
		for (int i = 0; i < v.undirectedDegree; i++) {
			Vertex neighbor = v.outNeighbors [i];
			if (! neighbor.booleanField) dft (neighbor);
		}
	}
}
```

```
package jigglecore;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class HybridVertexVertexRepulsionLaw extends VertexVertexRepulsionLaw {

        public HybridVertexVertexRepulsionLaw (Graph g, double k) {
                super (g, k);
        }

        double pairwiseRepulsion (Cell c1, Cell c2) {
                double r = Cell.sumOfRadii (c1, c2);
                double k = preferredEdgeLength + r;
                double dSquared = Cell.getDistanceSquared (c1, c2);
                if (dSquared < k * k) return k * k / dSquared;
                else return cube (k / Cell.getDistance (c1, c2));
        }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package jigglecore;

/**
 *
 * @author Bilal Khan
 */
public class Intersector {

public static int crossingNumber(Graph g) {
    int crossings = 0;

    // System.out.println("DEBUG crossingNumber over numberOfEdges="+g.numberOfEdges);

    for (int i=0; i<g.numberOfEdges; i++) {
        for (int j=i+1; j<g.numberOfEdges; j++) {
            if (intersect(g.edges[i], g.edges[j])) {
                crossings++;
            }
        }
    }
    return crossings;
}


static boolean intersect(Edge this_line, Edge other_line) {

    double Ax=this_line.getFrom().getCoords()[0];
    double Ay=this_line.getFrom().getCoords()[1];
    double Bx=this_line.getTo().getCoords()[0];
    double By=this_line.getTo().getCoords()[1];

    double Cx=other_line.getFrom().getCoords()[0];
    double Cy=other_line.getFrom().getCoords()[1];
    double Dx=other_line.getTo().getCoords()[0];
    double Dy=other_line.getTo().getCoords()[1];

    boolean xxx = intersect(Ax, Ay, Bx, By, Cx, Cy, Dx, Dy);

    /*
    System.out.print("("+Ax+","+Ay+")-");
    System.out.print("("+Bx+","+By+") and ");
    System.out.print("("+Cx+","+Cy+")-");
```

```
    System.out.print("("+Dx+","+Dy+")-");
    if (xxx) System.out.print(" INTERSECT");
    else System.out.println(" dont intersect");
    */

    return xxx;

}
static boolean intersect(double Ax, double Ay,
        double Bx, double By,
        double Cx, double Cy,
        double Dx, double Dy) {

    double  distAB, theCos, theSin, newX, ABpos ;

  //  Fail if either line segment is zero-length.
  if (Ax==Bx && Ay==By || Cx==Dx && Cy==Dy) return false;

  //  Fail if the segments share an end-point.
  if (Ax==Cx && Ay==Cy || Bx==Cx && By==Cy
  ||  Ax==Dx && Ay==Dy || Bx==Dx && By==Dy) {
    return false; }

  //  (1) Translate the system so that point A is on the origin.
  Bx-=Ax; By-=Ay;
  Cx-=Ax; Cy-=Ay;
  Dx-=Ax; Dy-=Ay;

  //  Discover the length of segment A-B.
  distAB=Math.sqrt(Bx*Bx+By*By);

  //  (2) Rotate the system so that point B is on the positive X axis.
  theCos=Bx/distAB;
  theSin=By/distAB;
  newX=Cx*theCos+Cy*theSin;
  Cy  =Cy*theCos-Cx*theSin; Cx=newX;
  newX=Dx*theCos+Dy*theSin;
  Dy  =Dy*theCos-Dx*theSin; Dx=newX;

  //  Fail if segment C-D doesn’t cross line A-B.
  if (Cy<0. && Dy<0. || Cy>=0. && Dy>=0.) return false;

  //  (3) Discover the position of the intersection point along line A-B.
  ABpos=Dx+(Cx-Dx)*Dy/(Dy-Cy);

  //  Fail if segment C-D crosses line A-B outside of segment A-B.
  if (ABpos<0. || ABpos>distAB) return false;
```

```
  //  Success.
  return true; }
}
```

```
package jigglecore;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class InverseSquareVertexEdgeRepulsionLaw extends VertexEdgeRepulsionLaw {

	public InverseSquareVertexEdgeRepulsionLaw (Graph g, double k) {
		super (g, k, 1);
	}

	public InverseSquareVertexEdgeRepulsionLaw (Graph g, double k, double s) {
		super (g, k, s);
	}

	double pairwiseRepulsion (Cell c1, Cell c2) {
		double k = preferredEdgeLength + Cell.sumOfRadii (c1, c2);
		double d = Cell.getDistance (c1, c2);
		if (d >= k) return 0; else return cube (k / d) - k / d;
	}

	double pairwiseRepulsion (Cell cell, double [] coords) {
		double k = preferredEdgeLength + Cell.radius (cell, coords);
		double d = Cell.getDistance (cell, coords);
		if (d >= k) return 0; else return cube (k / d) - k / d;
	}
}
```

```java
package jigglecore;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class InverseSquareVertexVertexRepulsionLaw extends VertexVertexRepulsionLaw {

	public InverseSquareVertexVertexRepulsionLaw (Graph g, double k) {
		super (g, k);
	}

	double pairwiseRepulsion (Cell c1, Cell c2) {
		double k = preferredEdgeLength + Cell.sumOfRadii (c1, c2);
		return cube (k / Cell.getDistance (c1, c2));
	}
}
```

```
package jigglecore;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class InverseVertexEdgeRepulsionLaw extends VertexEdgeRepulsionLaw {

        public InverseVertexEdgeRepulsionLaw (Graph g, double k) {
                super (g, k, 1);
        }

        public InverseVertexEdgeRepulsionLaw (Graph g, double k, double s) {
                super (g, k, s);
        }

        double pairwiseRepulsion (Cell c1, Cell c2) {
                double k = preferredEdgeLength + Cell.sumOfRadii (c1, c2);
                double dSquared = Cell.getDistanceSquared (c1, c2);
                if (dSquared >= square (k)) return 0;
                else return k * k / dSquared - k / Math.sqrt (dSquared);
        }

        double pairwiseRepulsion (Cell cell, double [] coords) {
                double k = preferredEdgeLength + Cell.radius (cell, coords);
                double dSquared = Cell.getDistanceSquared (cell, coords);
                if (dSquared >= square (k)) return 0;
                else return k * k / dSquared - k / Math.sqrt (dSquared);
        }
}
```

```
package jigglecore;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class InverseVertexVertexRepulsionLaw extends VertexVertexRepulsionLaw {

	public InverseVertexVertexRepulsionLaw (Graph g, double k) {
		super (g, k);
	}

	double pairwiseRepulsion (Cell c1, Cell c2) {
		double k = preferredEdgeLength + Cell.sumOfRadii (c1, c2);
		return k * k / Cell.getDistanceSquared (c1, c2);
	}
}
```

```
package jigglecore;

/* Abstract base class for all JIGGLE objects. */

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public abstract class JiggleObject {

        private JiggleObject context = null;
        public JiggleObject getContext () {return context;}
        protected void setContext (JiggleObject c) {context = c;}

        /* The context of a JiggleObject identifies the parent JiggleObject
        (if any) that contains it.  The context of a Vertex or Cell is either
        a Graph or a Cell; the context of an Edge is a Graph; the context of
        an EdgeLabel is an Edge.  For now, we assume that the   context of a
        Graph is null; if, however, we extend the present implementation to
        include composite graphs, then the context of a Graph   could be a
        JiggleObject (e.g. a Vertex) that contains the graph inside     it. */

        boolean booleanField = false;
        int intField = 0;
        Object objectField = null;

        static double square (double d) {return d * d;}
        static double cube (double d) {return d * d * d;}
        static int intSquare (int n) {return n * n;}

        static int power (int base, int d) {
                if (d == 0) return 1;
                else if (d == 1) return base;
                else if (d % 2 == 0) return intSquare (power (base, d / 2));
                else return base * intSquare (power (base, d / 2));
        }
}
```

```
package jigglecore;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class LinearSpringLaw extends SpringLaw {

        public LinearSpringLaw (Graph g, double k) {super (g, k);}

        double springAttraction (Edge e) {
                double r = Cell.sumOfRadii (e.getFrom (), e.getTo ());
                if (r == 0) return 1; else return 1 - r / e.getLength ();
        }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package jigglecore;

/**
 *
 * @author grouptheory
 */
public class Main {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

/*
Node 0 @ 5.999999999999999 , 0.0
Node 1 @ 2.8890725407747664 , 0.3191581928043395
Node 2 @ 4.1978494452756925 , 5.999999999999999
Node 3 @ 1.5365544482374358 , 2.793475452785374
Node 4 @ 0.0 , 0.8532927763978876
*/

        double Ax=2.8890725407747664;
        double Ay=0.3191581928043395;
        double Bx=4.1978494452756925;
        double By=5.999999999999999;

        double Cx=1.5365544482374358;
        double Cy=2.793475452785374;
        double Dx=5.999999999999999;
        double Dy=0.0;

        boolean inter = Intersector.intersect(Ax, Ay, Bx, By, Cx, Cy, Dx, Dy);

        System.out.println("The line intersect test: "+inter);
    }
}
```

```
package jigglecore;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class MultidimensionalArray extends JiggleObject {

	private int dimensions = 0;
	private int size [] = null;
	private int numberOfCells = 0;
	private Object cells [] = null;

	MultidimensionalArray (int d, int [] s) {
		dimensions = d; size = new int [d]; numberOfCells = 1;
		for (int i = 0; i < d; i++) {
			numberOfCells *= (size [i] = s [i]);
		}
		cells = new Object [numberOfCells];
	}

	int getDimensions () {return dimensions;}

	Object get (int [] index) {return cells [rankOf (index)];}

	void set (int [] index, Object obj) {cells [rankOf (index)] = obj;}

	private int rankOf (int [] index) {
		int rank = 0, column = 1;
		for (int i = 0; i < dimensions; i++) {
			rank += index [i] * column;
			column *= size [i];
		}
		return rank;
	}
}
```

```
package jigglecore;

/* Exception thrown by remove methods when element is not found. */

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class NotFoundException extends Exception {

        NotFoundException () {}
}
```

```java
package jigglecore;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class ProjectionConstraint extends Constraint {

	private int dimensions = 0;

	public ProjectionConstraint (Graph g, int d) {
		super (g); dimensions = d;
	}

	void apply (double [][] penalty) {
		int d = graph.getDimensions ();
		int n = graph.numberOfVertices;
		for (int i = 0; i < n; i++) {
			double coords [] = graph.vertices [i].getCoords ();
			for (int j = dimensions; j < d; j++) {
				penalty [i] [j] += (- coords [j]);
			}
		}
	}
}
```

```
package jigglecore;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class QuadTree extends Cell {

        QuadTree subtrees [];

        double force [];

        QuadTree (Graph g) {
                setContext (g); objectField = null;
                int d = g.getDimensions (); setDimensions (d);
                subtrees = new QuadTree [power (2, d)];
                int n = g.numberOfVertices;
                setMin (g.getMin ()); setMax (g.getMax ());
                for (int i = 0; i < n; i++)
                        g.vertices [i].objectField = null;
                for (int i = 0; i < n; i++)
                        insert (g.vertices [i]);
                force = new double [d]; for (int i = 0; i < d; i++) force [i] = 0;
        }

        private QuadTree (double [] min, double [] max, QuadTree p) {
                setContext (p); objectField = null;
                int d = p.getDimensions (); setDimensions (d);
                subtrees = new QuadTree [power (2, d)];
                setMin (min); setMax (max);
                force = new double [d]; for (int i = 0; i < d; i++) force [i] = 0;
        }

        QuadTree lookUp (Vertex v) {
                if (objectField == v) return this;
                else if (objectField != null) return null;
                else return subtrees [getIndex (v)].lookUp (v);
        }

        private int getIndex (Vertex v) {
                double c [] = v.getCoords (), center [] = getCenter ();
                int d = getDimensions (), index = 0, column = 1;
                for (int i = 0; i < d; i++) {
                        if (c [i] > center [i]) index += column;
                        column *= 2;
                }
                return index;
```

```
	}

	private double [] getCenter () {
		int d = getDimensions ();
		double mp [] = new double [d];
		double lo [] = getMin (), hi [] = getMax ();
		for (int i = 0; i < d; i++) mp [i] = (lo [i] + hi [i]) / 2;
		return mp;
	}

	protected void recomputeSize () {}
	void recomputeBoundaries () {}
	/* NOTE: Size for quadtrees has nothing to do with min and max! It stores
	the average size of the vertices that have been inserted into the tree. */

	void insert (Vertex v) {
		double w = getWeight (), vw = v.getWeight ();
		int d = getDimensions ();
		double vCoords [] = v.getCoords (), vSize [] = v.getSize ();
		if (w == 0) {
			v.setContext (this); setWeight (v.getWeight ());
			setCoords (vCoords); setSize (v.getSize ());
			objectField = v; return;
		}
		if (objectField != null) splitCell ();
		double c [] = getCoords (), s [] = getSize ();
		for (int i = 0; i < d; i++) {
			c [i] = (c [i] * w + vCoords [i] * vw) / (w + vw);
			s [i] = (s [i] * w + vSize [i] * vw) / (w + vw);
		}
		setWeight (w + vw);
		subtrees [getIndex (v)].insert (v);
	}

	private void splitCell () {
		Vertex v = (Vertex) objectField;
		objectField = null;
		double cellMin [] = getMin (), cellMax [] = getMax ();
		double center [] = getCenter ();
		int d = getDimensions (), n = power (2, d);
		double lo [] = new double [d], hi [] = new double [d];
		for (int index = 0; index < n; index++) {
			int column = 1;
			for (int i = 0; i < d; i++) {
				if ((index & column) > 0) {
					lo [i] = center [i]; hi [i] = cellMax [i];
				}
```

```
                                else {lo [i] = cellMin [i]; hi [i] = center [i];}
                                column *= 2;
                        }
                        subtrees [index] = new QuadTree (lo, hi, this);
                }
                subtrees [getIndex (v)].insert (v);
        }
}
```

```
package jigglecore;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class QuadraticSpringLaw extends SpringLaw {

	public QuadraticSpringLaw (Graph g, double k) {super (g, k);}

	double springAttraction (Edge e) {
		double r = Cell.sumOfRadii (e.getFrom (), e.getTo ());
		double len = e.getLength ();
		return (len - r) / preferredEdgeLength;
	}
}
```

```java
package jigglecore;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public abstract class SpringLaw extends ForceLaw {

	protected double preferredEdgeLength;

	protected SpringLaw (Graph g, double k) {
		super (g); preferredEdgeLength = k;
	}

	void apply (double [][] negativeGradient) {
		int m = graph.numberOfEdges, d = graph.getDimensions ();
		for (int i = 0; i < m; i++) {
			Edge e = graph.edges [i];
			Vertex from = e.getFrom (), to = e.getTo ();
			double fromWeight = from.getWeight (), toWeight = to.getWeight ();
			int f = from.intField, t = to.intField;
			double w = Math.min (springAttraction (e), cap / e.getLength ());
			double fromCoords [] = from.getCoords ();
			double toCoords [] = to.getCoords ();
			for (int j = 0; j < d; j++) {
				double force = (toCoords [j] - fromCoords [j]) * w;
				negativeGradient [f] [j] += force * toWeight;
				negativeGradient [t] [j] -= force * fromWeight;
			}
		}
	}

	abstract double springAttraction (Edge e);
}
```

```
package jigglecore;

/* Class for standard force model of graph-drawing aesthetics. */

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class StandardForceModel extends ForceModel {

        public StandardForceModel (Graph g, double k, double theta) {
                super (g);
                preferredEdgeLength = k;
                SpringLaw springLaw = new QuadraticSpringLaw (g, k);
                VertexVertexRepulsionLaw vvRepulsionLaw = new HybridVertexVertexRepulsionLaw (g, k);
                addForceLaw (springLaw);
                addForceLaw (vvRepulsionLaw);
                addConstraint (new ProjectionConstraint (g, 2));
        }
}
```

```
package jigglecore;

/* Class for method of steepest descent. */

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class SteepestDescent extends FirstOrderOptimizationProcedure{

	public SteepestDescent (Graph g, ForceModel fm, double accuracy) {
		super (g, fm, accuracy);
	}

	protected void computeDescentDirection () {
		int n = graph.numberOfVertices, d = graph.getDimensions ();
		if ((descentDirection == null) || (descentDirection.length != n))
			descentDirection = new double [n] [d];
		for (int i = 0; i < n; i++) {
			for (int j = 0; j < d; j++) {
				descentDirection [i] [j] = negativeGradient [i] [j];
			}
		}
	}
}
```

```
package jigglecore;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class SurfaceOfSphereConstraint extends Constraint {

	private double radius;

	public SurfaceOfSphereConstraint (Graph g) {
		super (g); radius = 0;
	}

	public SurfaceOfSphereConstraint (Graph g, double r) {
		super (g); radius = r;
	}

	void apply (double [][] penalty) {
		int d = graph.getDimensions ();
		int n = graph.numberOfVertices;
		double center [] = new double [d], sum [] = new double [d];
		for (int i = 0; i < d; i++) center [i] = sum [i] = 0;
		for (int i = 0; i < n; i++) {
			double coords [] = graph.vertices [i].getCoords ();
			for (int j = 0; j < d; j++) center [j] += coords [j] / n;
		}
		double r = radius;
		if (r == 0) {
			for (int i = 0; i < n; i++) {
				double coords [] = graph.vertices [i].getCoords ();
				double distanceSquared = 0;
				for (int j = 0; j < d; j++) {
					distanceSquared += square (coords [j] - center [j]);
				}
				r += Math.sqrt (distanceSquared);
			}
			r = r / n;
		}

		for (int i = 0; i < n; i++) {
			double coords [] = graph.vertices [i].getCoords ();
			double distanceSquared = 0;
			for (int j = 0; j < d; j++) {
				distanceSquared += square (coords [j] - center [j]);
			}
			double p = r - Math.sqrt (distanceSquared);
```

```
				for (int j = 0; j < d; j++) {
					penalty [i] [j] += p * (coords [j] - center [j]);
					sum [j] += p * (coords [j] - center [j]);
				}
			}
			for (int i = 0; i < n; i++) {
				for (int j = 0; j < d; j++) {
					penalty [i] [j] -= sum [j] / n;
				}
			}

		}
}
```

```
package jigglecore;

/* Class for vertices of a graph. */

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class Vertex extends Cell {

        int undirectedDegree = 0, inDegree = 0, outDegree = 0;
        Edge undirectedEdges [] = new Edge [1];
        Edge         inEdges [] = new Edge [1];
        Edge        outEdges [] = new Edge [1];
        Vertex undirectedNeighbors [] = new Vertex [1];
        Vertex         inNeighbors [] = new Vertex [1];
        Vertex        outNeighbors [] = new Vertex [1];

        /* NOTE: the above are made package-accessible for reasons of
        efficiency.  They should NOT, however, be modified except by
        insertNeighbor and deleteNeighbor methods below. */

        private String name = ""; /* name of vertex */
        private boolean fixed = false; /* is the vertex anchored? */

        Vertex (Graph g) {
                super ();
                setContext (g); setWeight (1); setDimensions (g.getDimensions ());
        }

        String getName () {return name;}
        void setName (String str) {name = str;}

        boolean getFixed () {return fixed;}
        void setFixed (boolean f) {fixed = f;}

        void insertNeighbor (Edge e) {
                Vertex from = e.getFrom (), to = e.getTo ();
                Vertex v = null;
                if (this == from) v = to; else if (this == to) v = from;
                else throw new Error (e + " not incident to " + this);
                if (! e.getDirected ()) {
                        undirectedEdges = DynamicArray.add
                                (undirectedEdges, undirectedDegree, e);
                        undirectedNeighbors = DynamicArray.add
                                (undirectedNeighbors, undirectedDegree++, v);
                }
```

```
			else if (this == from) {
				outEdges = DynamicArray.add (outEdges, outDegree, e);
				outNeighbors = DynamicArray.add
					(outNeighbors, outDegree++, to);
			}
			else {
				inEdges = DynamicArray.add (inEdges, inDegree, e);
				inNeighbors = DynamicArray.add
					(inNeighbors, inDegree++, from);
			}
	}

	void deleteNeighbor (Edge e) {
		Vertex from = e.getFrom (), to = e.getTo ();
		Vertex v = null;
		if (this == from) v = to; else if (this == to) v = from;
		else throw new Error (e + " not incident to " + this);
		try {
			if (! e.getDirected ()) {
				DynamicArray.remove
					(undirectedEdges, undirectedDegree, e);
				DynamicArray.remove
					(undirectedNeighbors, undirectedDegree--, v);
			}
			else if (this == from) {
				DynamicArray.remove (outEdges, outDegree, e);
				DynamicArray.remove (outNeighbors, outDegree--, to);
			}
			else {
				DynamicArray.remove (inEdges, inDegree, e);
				DynamicArray.remove (inNeighbors, inDegree--, from);
			}
		} catch (NotFoundException exc) {
			throw new Error (e + " not incident to " + this);
		}
	}

	public String toString () {return "(Vertex: " + name + ")";}
}
```

```
package jigglecore;

import java.util.Enumeration;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public abstract class VertexEdgeRepulsionLaw extends ForceLaw {

	protected double preferredEdgeLength;
	protected double strength = 1;

	protected VertexEdgeRepulsionLaw (Graph g, double k, double s) {
		super (g); preferredEdgeLength = k; strength = s;
	}

	private boolean gridding = false;
	public boolean getGridding () {return gridding;}
	public void setGridding (boolean b) {gridding = b;}

	abstract double pairwiseRepulsion (Cell c1, Cell c2);
	abstract double pairwiseRepulsion (Cell c, double [] coords);

	void apply (double [][] negativeGradient) {
		if (gridding) applyUsingGridding (negativeGradient);
		int n = graph.numberOfVertices, m = graph.numberOfEdges;
		int d = graph.getDimensions ();
		for (int i = 0; i < n; i++) {
			Vertex v = graph.vertices [i];
			for (int j = 0; j < m; j++) {
				Edge e = graph.edges [j];
				Vertex from = e.getFrom (), to = e.getTo ();
				computeRepulsion (v, e, negativeGradient);
			}
		}
	}

	private void applyUsingGridding (double [][] negativeGradient) {
		graph.recomputeBoundaries ();
		int n = graph.numberOfVertices, m = graph.numberOfEdges;
		int d = graph.getDimensions ();
		int gridSize [] = new int [d];
		double drawingArea [] = graph.getSize (), k = preferredEdgeLength;
		for (int i = 0; i < d; i++) {
			gridSize [i] = (int) (drawingArea [i] / k) + 1;
		}
```

```
        MultidimensionalArray grid = new MultidimensionalArray (d, gridSize);
        double gMin [] = graph.getMin ();
        int index [] = new int [d], sign [] = new int [d];
        for (int i = 0; i < n; i++) {
                Vertex v = graph.vertices [i]; double c [] = v.getCoords ();
                for (int j = 0; j < d; j++) {
                        index [j] = (int) ((c [j] - gMin [j]) / k);
                }
                VertexSet gridCell = (VertexSet) grid.get (index);
                if (gridCell == null) grid.set (index, new VertexSet (v));
                else gridCell.add (v);
                v.objectField = index;
        }
        for (int i = 0; i < m; i++) {
                Edge e = graph.edges [i];
                Vertex from = e.getFrom (), to = e.getTo ();
                double fCoords [] = from.getCoords (), tCoords [] = to.getCoords ();
                for (int j = 0; j < d; j++) {
                        if (fCoords [j] < tCoords [j]) sign [j] = 1;
                        else if (fCoords [j] > tCoords [j]) sign [j] = -1;
                        else sign [j] = 0;
                }
                int current [] = (int []) from.objectField;
                int numberOfAdjs = power (3, d);
                boolean flag = true;
                while (flag || (! equal (current, (int []) to.objectField))) {
                        flag = false;
                        FORLOOP: for (int adj = 0; adj < numberOfAdjs; adj++) {
                                int temp = adj;
                                for (int j = 0; j < d; j++) {
                                        index [j] = current [j] + (temp % 3) - 1;
                                        if ((index [j] < 0) || (index [j] >= gridSize [j]))
                                                continue FORLOOP;
                                        temp /= 3;
                                }
                                VertexSet gridCell = (VertexSet) grid.get (index);
                                if ((gridCell != null) && (! gridCell.booleanField)) {
                                        for (Enumeration en = gridCell.elements ();
                                             en.hasMoreElements ();) {
                                                Vertex v = (Vertex) en.nextElement ();
                                                computeRepulsion (v, e, negativeGradient);
                                        }
                                        gridCell.booleanField = true;
                                }
                        }
                        double time, minTime = Double.MAX_VALUE; int nextAxis = 0;
                        for (int axis = 0; axis < d; axis++) {
```

```
                                        if (sign [axis] == 0) continue;
                                        if (sign [axis] == 1) {
                                                time = (current [axis] + 1) * k /
                                                        (tCoords [axis] - fCoords [axis]);
                                        }
                                        else {
                                                time = current [axis] * k /
                                                        (fCoords [axis] - tCoords [axis]);
                                        }
                                        if (time < minTime) {minTime = time; nextAxis = axis;}
                                }
                                current [nextAxis] += sign [nextAxis];
                        }
                }
        }

        private boolean equal (int [] u, int [] v) {
                int d = u.length;
                for (int i = 0; i < d; i++) if (u [i] != v [i]) return false;
                return true;
        }

        private void computeRepulsion (Vertex v, Edge e, double [][] negativeGradient) {
                Vertex from = e.getFrom (), to = e.getTo ();
                if ((from == v) || (to == v)) return;
                int d = v.getDimensions ();
                double vCoords [] = v.getCoords ();
                double fCoords [] = from.getCoords (), tCoords [] = to.getCoords ();
                double dp = 0, lenSquared;
                for (int i = 0; i < d; i++) {
                        dp += (vCoords [i] - fCoords [i]) * (tCoords [i] - fCoords [i]);
                }
                if (dp <= 0) computeRepulsion (v, from, negativeGradient);
                else if (dp >= (lenSquared = e.getLengthSquared ()))
                        computeRepulsion (v, to, negativeGradient);
                else {
                        double len = Math.sqrt (lenSquared), alpha = dp / len;
                        double pCoords [] = new double [d];
                        for (int i = 0; i < d; i++) {
                                pCoords [i] = (1 - alpha) * fCoords [i] + alpha * tCoords [i];
                        }
                        double w = Math.min (strength * pairwiseRepulsion (v, pCoords),
                                             cap / Vertex.getDistance (v, pCoords));
                        if (w == 0) return;
                        double vWeight = v.getWeight ();
                        double fWeight = from.getWeight (), tWeight = to.getWeight ();
                        for (int i = 0; i < d; i++) {
```

```
                                double force1 = (vCoords [i] - fCoords [i]) * w * (1 - alpha);
                                double force2 = (vCoords [i] - tCoords [i]) * w * alpha;
                                negativeGradient [v.intField] [i] += force1 * fWeight;
                                negativeGradient [from.intField] [i] -= force1 * vWeight;
                                negativeGradient [v.intField] [i] += force2 * tWeight;
                                negativeGradient [to.intField] [i] -= force2 * vWeight;
                        }
                }
        }

        private void computeRepulsion (Vertex v1, Vertex v2, double [][] negativeGradient) {
                int d = v1.getDimensions ();
                double w = Math.min (strength * pairwiseRepulsion (v1, v2),
                                     cap / Vertex.getDistance (v1, v2));
                if (w == 0) return;
                double v1Coords [] = v1.getCoords (), weight1 = v1.getWeight ();
                double v2Coords [] = v2.getCoords (), weight2 = v2.getWeight ();
                for (int i = 0; i < d; i++) {
                        double force = (v1Coords [i] - v2Coords [i]) * w;
                        negativeGradient [v1.intField] [i] += force * weight2;
                        negativeGradient [v2.intField] [i] -= force * weight1;
                }
        }
}
```

```
package jigglecore;

import java.util.Vector;
import java.util.Enumeration;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public class VertexSet extends JiggleObject {

        private Vector vertices;

        VertexSet () {vertices = new Vector ();}

        VertexSet (Vertex v) {vertices = new Vector (); vertices.addElement (v);}

        void add (Vertex v) {vertices.addElement (v);}

        Enumeration elements () {return vertices.elements ();}
}
```

```
package jigglecore;

/**
 *
 * @author Daniel Tunkelang, minor edits by Bilal Khan
 */
public abstract class VertexVertexRepulsionLaw extends ForceLaw {

	protected double preferredEdgeLength;

	abstract double pairwiseRepulsion (Cell c1, Cell c2);

	private double barnesHutTheta = 0;

	protected VertexVertexRepulsionLaw (Graph g, double k) {
		super (g); preferredEdgeLength = k;
	}

	public double getBarnesHutTheta () {return barnesHutTheta;}
	public void setBarnesHutTheta (double t) {barnesHutTheta = t;}

	void apply (double [][] negativeGradient) {
		if (barnesHutTheta > 0)
			applyUsingBarnesHut (negativeGradient);
		int n = graph.numberOfVertices, d = graph.getDimensions ();
		for (int i = 0; i < n - 1; i++) {
			Vertex v1 = graph.vertices [i];
			double v1Coords [] = v1.getCoords (), weight1 = v1.getWeight ();
			for (int j = i + 1; j < n; j++) {
				Vertex v2 = graph.vertices [j];
				double w = Math.min (pairwiseRepulsion (v1, v2),
					cap / Vertex.getDistance (v1, v2));
				double v2Coords [] = v2.getCoords (), weight2 = v2.getWeight ();
				for (int k = 0; k < d; k++) {
					double force = (v1Coords [k] - v2Coords [k]) * w;
					negativeGradient [i] [k] += force * weight2;
					negativeGradient [j] [k] -= force * weight1;
				}
			}
		}
	}

	private void applyUsingBarnesHut (double [][] negativeGradient) {
		int n = graph.numberOfVertices, d = graph.getDimensions ();
		if (n <= 1) return;
		graph.recomputeBoundaries ();
		QuadTree root = new QuadTree (graph);
```

```
		for (int i = 0; i < n; i++) {
			Vertex v = graph.vertices [i];
			QuadTree qt = (QuadTree) v.getContext ();
			JiggleObject cur = qt;
			while (cur.getContext () != graph) {
				QuadTree p = (QuadTree) cur.getContext ();
				int numberOfSubtrees = power (2, d);
				for (int j = 0; j < numberOfSubtrees; j++) {
					QuadTree st = p.subtrees [j];
					if (cur != st) {
						computeQTRepulsion (qt, st, negativeGradient);
					}
				}
				cur = p;
			}
		}
		pushForcesDownTree (root);
		for (int i = 0; i < n; i++) {
			Vertex v = graph.vertices [i];
			QuadTree qt = (QuadTree) v.getContext ();
			for (int j = 0; j < d; j++) negativeGradient [i] [j] += qt.force [j];
			v.setContext (graph);
		}
	}

	private void computeQTRepulsion (QuadTree leaf, QuadTree cell, double [][] negativeGradient) {
		if (cell == null) return;
		int d = leaf.getDimensions ();
		if ((cell.objectField == null) && (! wellSeparated (leaf, cell))) {
			int numberOfSubtrees = power (2, d);
			for (int i = 0; i < numberOfSubtrees; i++) {
				computeQTRepulsion (leaf, cell.subtrees [i], negativeGradient);
			}
		}
		else {
			double w = Math.min (pairwiseRepulsion (leaf, cell),
			                     cap / Cell.getDistance (leaf, cell));
			double leafWeight = leaf.getWeight (), cellWeight = cell.getWeight ();
			double leafCoords [] = leaf.getCoords (), cellCoords [] = cell.getCoords ();
			int i = leaf.intField;
			for (int j = 0; j < d; j++) {
				double force = 0.5 * w * (leafCoords [j] - cellCoords [j]);
				negativeGradient [i] [j] += force * cellWeight;
				cell.force [j] -= force * leafWeight;
			}
		}
	}
```

```
	private boolean wellSeparated (QuadTree leaf, QuadTree cell) {
		if (cell == null) throw new Error ("cell == null");
		if (cell.objectField != null) return true;
		else {
			int d = cell.getDimensions ();
			double len = Double.MAX_VALUE;
			double lo [] = cell.getMin (), hi [] = cell.getMax ();
			for (int i = 0; i < d; i++) len = Math.min (len, hi [i] - lo [i]);
			double dist = Cell.getDistance (leaf, cell);
			return ((len / dist) < barnesHutTheta);
		}
	}

	private void pushForcesDownTree (QuadTree qt) {
		if ((qt != null) && (qt.objectField == null) && (qt.getWeight () > 0)) {
			int d = qt.getDimensions (), numberOfSubtrees = power (2, d);
			for (int i = 0; i < numberOfSubtrees; i++) {
				for (int j = 0; j < d; j++) {
					qt.subtrees [i].force [j] += qt.force [j];
				}
			}
			for (int i = 0; i < numberOfSubtrees; i++) {
				pushForcesDownTree (qt.subtrees [i]);
			}
		}
	}
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package letter;

/**
 *
 * @author grouptheory
 */

public class Constant extends Letter {

    public Boolean isConstant() {
        return true;
    }

    private Boolean _pos;

    public Boolean isPositive() {
        return _pos;
    }

    private Constant _inverse;

    public Letter getInverse() {
        if (_inverse == null) {
            throw new RuntimeException("Constant.getInverse: _inverse == null");
        }
        return _inverse;
    }

    Constant(int id, Boolean positive) {
        super(id);
        _pos = positive;
    }

    void setInverse(Constant cinv) {
        _inverse = cinv;
    }

    public String toString() {
        String s = "";
        s += CONSTANT_SYMBOL;
        s += _id;
        if (_pos) {
```

```
            s += POSITIVE_SYMBOL;
        }
        else {
            s += NEGATIVE_SYMBOL;
        }
        s += ENDOFLETTER_SYMBOL;
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package letter;

/**
 *
 * @author grouptheory
 */
public class Latex {

    private static Latex _instance;

    private Latex() {
    }

    public static Latex instance() {
        if (_instance == null) {
            _instance = new Latex();
        }
        return _instance;
    }

    public String render(Letter let) {
        String s = "";
        if (let.isConstant()) s+= this.renderConstant((Constant)let);
        else s+= this.renderVariable((Variable)let);
        return s;
    }

    String renderVariable(Variable v) {
        String s = "";
        s += "z_{";
        s += ""+v.getID()+"}";
        if ( ! v.isPositive()) {
            s += "^{-1}";
        }
        return s;
    }

    String renderConstant(Constant c) {
        String s = "";
        s += "c_";
        s += "{"+c.getID()+"}";
        if ( ! c.isPositive()) {
```

```
            s += "^{-1}";
        }
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package letter;

/**
 *
 * @author grouptheory
 */

public abstract class Letter {

    static String VARIABLE_SYMBOL = "z";
    static String CONSTANT_SYMBOL = "c";
    static String POSITIVE_SYMBOL = "+";
    static String NEGATIVE_SYMBOL = "-";
    static String ENDOFLETTER_SYMBOL = ".";

    public abstract Boolean isConstant();
    public abstract Boolean isPositive();
    public abstract Letter getInverse();

    protected int _id;

    protected Letter(int id) {
        _id = id;
    }

    public int getID() {
        return _id;
    }

    public int modulus(int mod) {
        return _id % mod;
    }

    public static Boolean testEquals(Letter a1, Letter a2) {
        return (a1==a2);
    }

    public static Boolean testInverse(Letter a1, Letter a2) {
        return (a1.getInverse()==a2);
    }

    public static Boolean testEqualOrInverse(Letter a1, Letter a2) {
```

```
        return (testEquals(a1,a2) || testInverse(a1,a2));
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package letter;

import java.util.TreeMap;
import java.util.TreeSet;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
public class LetterFactory {

    private TreeMap _id2constant_pos;
    private TreeMap _id2constant_neg;
    private TreeMap _id2variable_pos;
    private TreeMap _id2variable_neg;
    private int _highconstant;
    private int _highvariable;

    private static LetterFactory _instance;

    private LetterFactory() {
        _id2constant_pos = new TreeMap();
        _id2constant_neg = new TreeMap();
        _id2variable_pos = new TreeMap();
        _id2variable_neg = new TreeMap();
        _highconstant = 0;
        _highvariable = 0;
    }

    public static LetterFactory instance() {
        if (_instance == null) {
            _instance = new LetterFactory();
        }
        return _instance;
    }

    public LinkedList parse(String s) {
        LinkedList letters = new LinkedList();
        LetterReaderState lrs;
```

```
        do {
            lrs = readNextLetter(s);
            s = lrs._remaining;
            letters.addLast(lrs._extracted);
        }
        while (s.length() > 0);
        return letters;
    }

    public Variable newUnusedConstant(Boolean pos) {
        return getVariable(_highconstant+1, pos);
    }

    public Constant getConstant(int id, Boolean pos) {
        Constant c = null;
        if (pos) {
            c = (Constant)_id2constant_pos.get(new Integer(id));
        }
        else {
            c = (Constant)_id2constant_neg.get(new Integer(id));
        }

        if (c == null) {
            c = new Constant(id, pos);
            Constant cinv = new Constant(id, !pos);
            c.setInverse(cinv);
            cinv.setInverse(c);
            if (pos) {
                _id2constant_pos.put(new Integer(id), c);
                _id2constant_neg.put(new Integer(id), cinv);
            }
            else {
                _id2constant_neg.put(new Integer(id), c);
                _id2constant_pos.put(new Integer(id), cinv);
            }
            if (_highconstant < id) {
                _highconstant = id;
            }
        }
        return c;
    }

    public Variable newUnusedVariable(Boolean pos) {
        //System.out.println(" high is  "+_highvariable);
        return getVariable(_highvariable+1, pos);
    }
```

```
    private Variable newUnusedVariable(int minID, Boolean pos) {
        if (minID > _highvariable+1) {
            // System.out.println("setting high to "+(minID-1));
            _highvariable = minID-1;
        }

        return getVariable(_highvariable+1, pos);
    }

    public Variable newUnusedVariable(HashSet used, int minID, Boolean pos) {
        TreeMap srch;
        if (pos) {
            srch = _id2variable_pos;
        }
        else {
            srch = _id2variable_neg;
        }

        Variable answer = null;
        for (Iterator it=srch.values().iterator(); it.hasNext();) {
            Variable v = (Variable)it.next();
            if (used.contains(v) || used.contains(v.getInverse())) continue;
            else {
                if (v.getID() < minID) continue;
                else {
                    answer = v;
                    break;
                }
            }
        }

        /*
        if (minID==0) {
            System.out.println("LetterFactory used set has size: "+used.size());
            for (Iterator it=used.iterator(); it.hasNext();) {
                Letter let = (Letter)it.next();
                System.out.println("used: "+let);
            }
            System.out.println("LetterFactory _highvariable: "+_highvariable);
            if (answer==null) {
                System.out.println("LetterFactory (answer==null)");
            }
        }
        */

        if (answer==null) {
            answer = newUnusedVariable(minID, pos);
```

```
        }

        /*
        if (minID==0) {
            System.out.println("POST LetterFactory used set has size: "+used.size());
            for (Iterator it=used.iterator(); it.hasNext();) {
                Letter let = (Letter)it.next();
                System.out.println("POST used: "+let);
            }
            System.out.println("POST LetterFactory _highvariable: "+_highvariable);
            if (answer==null) {
                System.out.println("POST LetterFactory (answer==null)");
            }
            System.out.println("POST LetterFactory returning: "+answer);
        }
        */

        return answer;
    }

    public Variable getVariable(int id, Boolean pos) {
        Variable v = null;
        if (pos) {
            v = (Variable)_id2variable_pos.get(new Integer(id));
        }
        else {
            v = (Variable)_id2variable_neg.get(new Integer(id));
        }

        if (v == null) {
            v = new Variable(id, pos);
            Variable vinv = new Variable(id, !pos);
            v.setInverse(vinv);
            vinv.setInverse(v);
            if (pos) {
                _id2variable_pos.put(new Integer(id), v);
                _id2variable_neg.put(new Integer(id), vinv);
            }
            else {
                _id2variable_neg.put(new Integer(id), v);
                _id2variable_pos.put(new Integer(id), vinv);
            }
            if (_highvariable < id) {
                // System.out.println("*** setting high to "+id);
                _highvariable = id;
            }
        }
```

```java
        return v;
    }


    private class LetterReaderState {
        public String _remaining;
        public Letter _extracted;
    }

    private LetterReaderState readNextLetter(String s) {

        //System.out.println("called w: "+s);

        Boolean var = false;
        if (s.charAt(0) == Letter.VARIABLE_SYMBOL.charAt(0)) {
            //System.out.println("var");
            var = true;
        }
        else if (s.charAt(0) == Letter.CONSTANT_SYMBOL.charAt(0)) {
            //System.out.println("const");
            var = false;
        }
        else {
            throw new RuntimeException("Variable.ctor: bad type");
        }

        int endofletter = s.indexOf(Letter.ENDOFLETTER_SYMBOL);
        //System.out.println("endofletter="+endofletter);

        String idstr = s.substring(1, endofletter-1);
        //System.out.println("idstr="+idstr);

        int id = Integer.parseInt(idstr);

        String posstr = s.substring(endofletter-1, endofletter);
        Boolean pos = false;
        if (posstr.compareTo(Letter.POSITIVE_SYMBOL)==0) {
            pos = true;
        }
        else if (posstr.compareTo(Letter.NEGATIVE_SYMBOL)==0) {
            pos = false;
        }
        else {
            throw new RuntimeException("Variable.ctor: bad pos/neg");
        }

        Letter let;
```

```
        if (var) {
            let = getVariable(id, pos);
        }
        else {
            let = getConstant(id, pos);
        }

        LetterReaderState lrs = new LetterReaderState();
        lrs._extracted = let;
        lrs._remaining = s.substring(endofletter+1);

        return lrs;
    }

    public String toString() {
        String s = "";
        s+="_id2constant_pos size = "+_id2constant_pos.size()+"\n";
        s+="_id2constant_neg size = "+_id2constant_neg.size()+"\n";
        s+="_id2variable_pos size = "+_id2variable_pos.size()+"\n";
        s+="_id2variable_neg size = "+_id2variable_neg.size()+"\n";
        return s;

    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package letter;

import java.util.LinkedList;
import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

        Letter z1 = LetterFactory.instance().getVariable(1, true);
        Letter z2 = LetterFactory.instance().getVariable(2, true);
        Letter z1inv = LetterFactory.instance().getVariable(1, false);
        Letter z2inv = LetterFactory.instance().getVariable(2, false);

        Letter c1 = LetterFactory.instance().getConstant(1, true);
        Letter c2 = LetterFactory.instance().getConstant(2, true);
        Letter c1inv = LetterFactory.instance().getConstant(1, false);
        Letter c2inv = LetterFactory.instance().getConstant(2, false);

        String s = "";
        s+=z1.toString();
        s+=z2.toString();
        s+=z1inv.toString();
        s+=z2inv.toString();
        s+=c1.toString();
        s+=c2.toString();
        s+=c1inv.toString();
        s+=c2inv.toString();
        s+=z1.toString();
        s+=z2.toString();
        s+=z1inv.toString();
        s+=z2inv.toString();
        s+=c1.toString();
        s+=c2.toString();
```

```
        s+=c1inv.toString();
        s+=c2inv.toString();

        System.out.println(s);

        System.out.println("Parsing string");

        LinkedList list = LetterFactory.instance().parse(s);
        for (Iterator it = list.iterator(); it.hasNext(); ) {
            Letter let = (Letter)it.next();
            System.out.println(let.toString());
        }

        System.out.println("LetterFactory: ");
        System.out.println(LetterFactory.instance().toString());
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package letter;

/**
 *
 * @author grouptheory
 */

public class Variable extends Letter {

    public Boolean isConstant() {
        return false;
    }

    private Boolean _pos;

    public Boolean isPositive() {
        return _pos;
    }

    private Variable _inverse;

    public Letter getInverse() {
        if (_inverse == null) {
            throw new RuntimeException("Variable.getInverse: _inverse == null");
        }
        return _inverse;
    }

    Variable(int id, Boolean positive) {
        super(id);
        _pos = positive;
    }

    void setInverse(Variable vinv) {
        _inverse = vinv;
    }

    public String toString() {
        String s = "";
        s += VARIABLE_SYMBOL;
        s += _id;
        if (_pos) {
```

```
            s += POSITIVE_SYMBOL;
        }
        else {
            s += NEGATIVE_SYMBOL;
        }
        s += ENDOFLETTER_SYMBOL;
        return s;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */


package makanin;

import ge.Base;

/**
 *
 * @author grouptheory
 */
public class BaseClassCarrier extends BaseClassDecorator {

    BaseClassCarrier() {
        super();
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.Base;
import ge.BaseDecorator;

/**
 *
 * @author grouptheory
 */

public abstract class BaseClassDecorator extends BaseDecorator {

    static final String NAME = "BaseClass";

    BaseClassDecorator() {
        super();
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.Base;
import ge.Boundary;
import ge.GE;
import java.util.Iterator;
import java.util.HashMap;

/**
 *
 * @author grouptheory
 */
public class BaseClassDecoratorFactory {

    private HashMap _base2class;

    public static void applyToAllBases(GE geq) {
        BaseClassDecoratorFactory bcf = new BaseClassDecoratorFactory(geq);
        for (Iterator it=geq.iteratorBases(); it.hasNext();) {
            Base bs=(Base)it.next();
            BaseClassDecorator bcd = bcf.newBaseClassDecorator(bs);
            bcd.attach(BaseClassDecorator.NAME, bs);
        }
    }

    private BaseClassDecoratorFactory(GE geq) {

        Carrier ca = (Carrier)geq.lookupDecorator(Carrier.NAME);
        if (ca == null) {
            throw new RuntimeException("BaseClassDecoratorFactory.ctor: unknown carrier");
        }
        Base carrier_base = ca.getBase();
        if (carrier_base == null) {
            throw new RuntimeException("BaseClassDecoratorFactory.ctor: carrier_base is null");
        }

        CriticalBoundary cr = (CriticalBoundary)geq.lookupDecorator(CriticalBoundary.NAME);
        if (cr == null) {
            throw new RuntimeException("BaseClassDecoratorFactory.ctor: unknown critical boundary");
        }
        Boundary critical_boundary = cr.getBoundary();
        if (critical_boundary == null) {
```

```
            throw new RuntimeException("BaseClassDecoratorFactory.ctor: critical_boundary is null");
        }

        _base2class = new HashMap();

        BaseClassDecorator abcCarrier = new BaseClassCarrier();
        _base2class.put(carrier_base, abcCarrier);

        for (Iterator it=geq.iteratorBases(); it.hasNext();) {
            Base bs=(Base)it.next();
            if (bs==carrier_base) continue;

            if (bs.isEmpty() && bs.getBegin().getID()<carrier_base.getBegin().getID()) {
                BaseClassDecorator abc = new BaseClassSuperfluous();
                _base2class.put(bs, abc);
            }
            else if(carrier_base.getBegin().getID() <= bs.getBegin().getID() &&
                bs.getBegin().getID() < critical_boundary.getID()) {
                BaseClassDecorator abc = new BaseClassTransport();
                _base2class.put(bs, abc);
            }
            else {
                BaseClassDecorator abc = new BaseClassFixed();
                _base2class.put(bs, abc);
            }
        }
    }

    private BaseClassDecorator newBaseClassDecorator(Base bs) {
        return (BaseClassDecorator)_base2class.get(bs);
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.Base;

/**
 *
 * @author grouptheory
 */
public class BaseClassFixed extends BaseClassDecorator {

    BaseClassFixed() {
        super();
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.Base;

/**
 *
 * @author grouptheory
 */
public class BaseClassSuperfluous extends BaseClassDecorator {

    BaseClassSuperfluous() {
        super();
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.Base;

/**
 *
 * @author grouptheory
 */
public class BaseClassTransport extends BaseClassDecorator {

    BaseClassTransport() {
        super();
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.Base;
import ge.GE;
import ge.GEDecorator;

/**
 *
 * @author grouptheory
 */
public class Carrier extends GEDecorator {

    static final String NAME = "Carrier";

    private Base _carrier;

    Carrier(Base carrier) {
        super();
        _carrier = carrier;
    }

    Base getBase() {
        return _carrier;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.Base;
import ge.GE;
import java.util.TreeSet;
import java.util.Iterator;
import ge.BaseComparator;

/**
 *
 * @author grouptheory
 */
public class CarrierFactory {

    public static void applyToGE(GE geq) {
        Carrier ca = CarrierFactory.compute(geq);
        ca.attach(Carrier.NAME, geq);
    }

    private static Carrier compute(GE geq) {
        TreeSet ts = new TreeSet(new BaseComparator());
        for (Iterator it=geq.iteratorBases(); it.hasNext();) {
            Base bs=(Base)it.next();
            ts.add(bs);
        }
        Base cb = (Base)ts.first();

        Carrier carrier = new Carrier(cb);
        return carrier;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import utility.AbstractComposableIterator;
import utility.ComposableIterator;
import java.util.LinkedList;
import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
public class ComposablePrintNodeIterator
        extends AbstractComposableIterator
        implements ComposableIterator {

    private LinkedList _substates;
    private Iterator _readi;
    private Iterator _childi;


    ComposablePrintNodeIterator(PrintNode pn, boolean root) {
        _substates = new LinkedList();
        if (pn.remainingCarrierBoundaries()>0) {
            _substates.addLast(pn.consumeCarrierBoundary(PrintNode.ZERO));
            if (!root) _substates.addLast(pn.consumeCarrierBoundary(PrintNode.NONZERO));
        }
        if (pn.remainingCarrierDualBoundaries()>0) {
            _substates.addLast(pn.consumeCarrierDualBoundary(PrintNode.ZERO));
            if (!root) _substates.addLast(pn.consumeCarrierDualBoundary(PrintNode.NONZERO));
        }

        _readi = _substates.iterator();
        _childi = _substates.iterator();
    }

    public ComposableIterator newComposableIterator(ComposableIterator parent) {
        PrintNode pn = (PrintNode)_childi.next();
        boolean root = false;
        return new ComposablePrintNodeIterator(pn, root);
    }

    public boolean hasNext() {
```

```
        return _readi.hasNext();
    }

    public Object next() {
        PrintNode answer = (PrintNode)_readi.next();
        return answer;
    }

    public void remove() {
        throw new RuntimeException("ComposablePrintNodeIterator.remove: not implemented");
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.Boundary;
import ge.GE;
import ge.GEDecorator;

/**
 *
 * @author grouptheory
 */
public class CriticalBoundary extends GEDecorator {

    static final String NAME = "CriticalBoundary";

    private Boundary _cr;

    CriticalBoundary(Boundary cr) {
        super();
        _cr = cr;
    }

    Boundary getBoundary() {
        return _cr;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.Base;
import ge.Boundary;
import ge.GE;
import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
public class CriticalBoundaryFactory {

    public static void applyToGE(GE geq) {
        Carrier ca = (Carrier)geq.lookupDecorator(Carrier.NAME);
        if (ca == null) {
            throw new RuntimeException("CriticalBoundarySelector.applyToGE: unknown carrier");
        }

        CriticalBoundary cb = CriticalBoundaryFactory.compute(geq, ca);
        cb.attach(CriticalBoundary.NAME, geq);
    }

    private static CriticalBoundary compute(GE geq, Carrier carrier) {
        Base cbs = carrier.getBase();

        Boundary cr = cbs.getEnd();

        for (Iterator it=geq.iteratorBases(); it.hasNext();) {
            Base bs=(Base)it.next();
            if (bs == cbs) continue;
            // find all bs with property that end(carrier) in col(bs)
            if (bs.getBegin().getID() < cbs.getEnd().getID() &&
                bs.getEnd().getID() > cbs.getEnd().getID()) {
                if (cr == null) {
                    cr = bs.getBegin();
                }
                else {
                    // find bs with minimal begin(bs) that has the above property
                    if (cr.getID() > bs.getBegin().getID()) {
                        cr = bs.getBegin();
                    }
```

```
                }
            }
        }

        CriticalBoundary cbd = new CriticalBoundary(cr);
        return cbd;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.GE;
import ge.Base;
import ge.Boundary;
import ge.GEDegeneracyTester;

/**
 *
 * @author grouptheory
 */
public class Latex {

    private static Latex _instance;

    private Latex() {
    }

    public static Latex instance() {
        if (_instance == null) {
            _instance = new Latex();
        }
        return _instance;
    }

    public String renderGEasText(GE geq) {
        String s = "";

        Carrier ca = (Carrier)geq.lookupDecorator(Carrier.NAME);
        if (ca == null) {
            throw new RuntimeException("Latex.renderGEasText: unknown carrier");
        }
        Base carrier_base = ca.getBase();
        if (carrier_base == null) {
            throw new RuntimeException("Latex.renderGEasText: carrier_base is null");
        }

        CriticalBoundary cr = (CriticalBoundary)geq.lookupDecorator(CriticalBoundary.NAME);
        if (cr == null) {
            throw new RuntimeException("Latex.renderGEasText: unknown critical boundary");
        }
        Boundary critical_boundary = cr.getBoundary();
```

```
        if (critical_boundary == null) {
            throw new RuntimeException("gutiLatexerrez.renderGEasText: critical_boundary is null");
        }

        Base carrier_dual = carrier_base.getDual();

        s+=("{\\bf Carrier Information}\n");
        s+="\\begin{verbatim}\n";
        s+=("Carrier: "+carrier_base+"\n");
        s+=("Carrier Dual: "+carrier_dual+"\n");
        s+=("Critical Boundary: "+critical_boundary+"\n");
        s+="\\end{verbatim}\n";

        return s;
    }

    public String renderPrintsasText(GE geq) {
        String s = "";

        s+=("{\\bf Prints}\n");
        s+="\\begin{verbatim}\n";
        int j=0;
        for (PrintIterator pi = PrintIteratorFactory.instance().newPrintIterator(geq);
             pi.hasNext();) {
            Print p = (Print)pi.next();
            s+=("     Print "+j+": "+p+"\n");
            j++;
        }
        s+="\\end{verbatim}\n";
        s+=("Total number of prints: "+j+"\\\\\n");

        j=0;
        for (PrintIterator pi = PrintIteratorFactory.instance().newPrintIterator(geq);
             pi.hasNext();) {
            Print p = (Print)pi.next();

            if (j==0) {
                s+=("{\\em First, we consider}\n");
            }
            else {
                s+=("{\\em Next, we consider}\n");
            }
            s+="\\begin{verbatim}\n";
            s+=("     Print "+j+": "+p+"\n");
            s+="\\end{verbatim}\n";

            PrintApplicator pa = new PrintApplicator(geq, p);
```

```
            GE childeq = pa.getPrinted();

            s+=("{\\bf Sequence of Actions in performing the Print "+j+":}\\\\\n");
            s+=pa.toString();

            s+=("{\\em Summarizing, the GE we obtain after applying}\n");
            s+="\\begin{verbatim}\n";
            s+=("     Print "+j+": "+p+"\n");
            s+="\\end{verbatim}\n";
            s+=("{\\em is shown below:}\n");

            s+=ge.Latex.instance().renderGEasGraphics(childeq);

            GEDegeneracyTester testchild = new GEDegeneracyTester(childeq);
            if (testchild.isDegenerate()) {
                s+=("Observe the following facts about this GE:\n");
                s+=testchild.toString();
                s+=("These observations show that the GE above is degenerate.\\\\[0.2in]\n");
            }
            else {
                s+=("The GE above is non-degenerate.\\\\[0.2in]\n");
            }
            s+=("This completes the consideration of Print "+j+".\\\\[0.2in]\n");

            j++;
        }

        return s;
    }

    public String renderGEasGraphics(GE geq) {
        String s = "";
        s += ge.Latex.instance().renderGEasGraphics(geq);
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import equation.GroupEquation;
import ge.TopLevelGEIterator;
import ge.TopLevelGEIteratorFactory;
import ge.GE;
import ge.Base;
import ge.Boundary;
import java.util.Iterator;
import utility.CompositeIterator;
import utility.CompositeIterator.State;

/**
 *
 * @author grouptheory
 */
public class Main {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

        GroupEquation problem = new GroupEquation("z1+.c1+.z2+.c2+.z1-.c3+.z2-.");

        TopLevelGEIterator iter = TopLevelGEIteratorFactory.instance().newTopLevelGEIterator(problem);

        int i=0;
        for (;iter.hasNext();) {
            GE geq = (GE)iter.next();

            CarrierFactory.applyToGE(geq);
            CriticalBoundaryFactory.applyToGE(geq);
            BaseClassDecoratorFactory.applyToAllBases(geq);

            Carrier ca = (Carrier)geq.lookupDecorator(Carrier.NAME);
            if (ca == null) {
                throw new RuntimeException("gutierrez.Main: unknown carrier");
            }
            Base carrier_base = ca.getBase();
            if (carrier_base == null) {
                throw new RuntimeException("gutierrez.Main: carrier_base is null");
```

```
            }

            CriticalBoundary cr = (CriticalBoundary)geq.lookupDecorator(CriticalBoundary.NAME);
            if (cr == null) {
                throw new RuntimeException("gutierrez.Main: unknown critical boundary");
            }
            Boundary critical_boundary = cr.getBoundary();
            if (critical_boundary == null) {
                throw new RuntimeException("gutierrez.Main: critical_boundary is null");
            }

            Base carrier_dual = carrier_base.getDual();

            System.out.println("GE#"+i+":\n");

            /*
            System.out.println(""+geq+"\n");
            System.out.println("\n");
            System.out.println("Carrier: "+carrier_base+"\n");
            System.out.println("Carrier Dual: "+carrier_dual);
            System.out.println("Critical Boundary: "+critical_boundary+"\n");
            System.out.println("\n\n");
            */

            i++;

            int j=0;
            for (PrintIterator pi = PrintIteratorFactory.instance().newPrintIterator(geq);
                 pi.hasNext();) {
                Print p = (Print)pi.next();
                // System.out.println("PRINT "+p);
                j++;
            }
            System.out.println("Total number of prints generated: "+j+"\n");

            //break;
        }

        System.out.println("Total number of GEs generated: "+i+"\n");
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.Boundary;
import java.util.Iterator;
import java.util.LinkedList;
/**
 *
 * @author grouptheory
 */
public class Print {

    private LinkedList _nodes;
    private boolean _flipped;

    Print(boolean flipped) {
        _nodes = new LinkedList();
        _flipped = flipped;
    }

    void append(PrintNode pn) {
        _nodes.addLast(pn);
    }

    PrintNode nextPrintNode(PrintNode pn) {
        int idx = _nodes.indexOf(pn);
        if (idx == _nodes.size()-1) {
            return null;
        }
        else {
            return (PrintNode)_nodes.get(idx+1);
        }
    }


    PrintNode prevPrintNode(PrintNode pn) {
        int idx = _nodes.indexOf(pn);
        if (idx == 0) {
            return null;
        }
        else {
            return (PrintNode)_nodes.get(idx-1);
        }
```

```java
    }

    public String toString() {
        String s="";
        for (Iterator it=_nodes.iterator();it.hasNext();) {
            PrintNode pn = (PrintNode)it.next();
            s += pn.toString();
        }
        return s;
    }

    public Iterator iteratorPrintNodes(boolean ascending) {
        if (ascending) {
            return _nodes.iterator();
        }
        else {
            return _nodes.descendingIterator();
        }
    }

    PrintNode get(Boundary b1, PrintNode.Source s1) {
        for (Iterator it=_nodes.iterator();it.hasNext();) {
            PrintNode pn = (PrintNode)it.next();
            if ((pn.getSource() == s1) && (pn.getBoundary()==b1)) {
                return pn;
            }
        }
        return null;
    }

    boolean contains(Boundary b1, PrintNode.Source s1) {
        return get(b1,s1)!=null;
    }

    Boundary getBegin() {
        Iterator it = (!_flipped) ? _nodes.iterator() : _nodes.descendingIterator();
        for (;it.hasNext();) {
            PrintNode pn = (PrintNode)it.next();
            if (pn.getSource() == PrintNode.DUAL) {
                return pn.getBoundary();
            }
        }
        return null;
    }

    Boundary getEnd() {
        Iterator it = (!_flipped) ? _nodes.descendingIterator() : _nodes.iterator() ;
```

```
        for (;it.hasNext();) {
            PrintNode pn = (PrintNode)it.next();
            if (pn.getSource() == PrintNode.DUAL) {
                return pn.getBoundary();
            }
        }
        return null;
    }

    int compare(Boundary b1, PrintNode.Source s1, Boundary b2, PrintNode.Source s2) {

        PrintNode p1 = null;
        PrintNode p2 = null;
        PrintNode first = null;
        for (Iterator it=_nodes.iterator();it.hasNext();) {
            PrintNode pn = (PrintNode)it.next();
            if ((pn.getSource() == s1) && (pn.getBoundary()==b1)) {
                p1 = pn;
                if (first==null) first=p1;
            }
            if ((pn.getSource() == s2) && (pn.getBoundary()==b2)) {
                p2 = pn;
                if (first==null) first=p2;
            }
            if (p1!=null && p2!=null) {
                break;
            }
        }

        // System.out.println("Comparing "+b1+":"+s1.toString()+" with "+b2+":"+s2.toString());

        if (p1==null) {
            throw new RuntimeException("Print.compare: boundary "+b1+" not found in "+s1);
        }

        if (p2==null) {
            throw new RuntimeException("Print.compare: boundary "+b2+" not found in "+s2);
        }

        int i1=_nodes.indexOf(p1);
        int i2=_nodes.indexOf(p2);
        int low,high;
        if (first==p1) {
            low=i1; high=i2;
        }
        else {
            low=i2; high=i1;
```

```
        }

        int distance = 0;
        for (int i=low+1; i<=high; i++) {
            PrintNode pn = (PrintNode)_nodes.get(i);
            if (pn.getOffset() == PrintNode.NONZERO) {
                distance++;
            }
        }

        distance *= ((first==p1) ? -1 : +1);

        return distance;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.GE;
import ge.Constraint;
import ge.Boundary;
import ge.Base;
import ge.IGETask;
import ge.GETask_ConstraintDeletion;
import ge.GETask_ConstraintAddition;
import ge.GETask_BoundaryInsertion;
import ge.GETask_MoveBase;
import ge.GETask_CollapseBase;
import ge.GETask_DeleteBase;
import ge.GETask_DeleteBoundary;
import java.util.Iterator;
import java.util.TreeMap;
import java.util.HashMap;
import java.util.LinkedList;

/**
 *
 * @author grouptheory
 */
public class PrintApplicator {

    private TreeMap _old2new_bdmap;
    private HashMap _old2new_bsmap;
    private GE _geqOriginal;
    private GE _geqPrinted;

    private TreeMap _crtr2new_bdmap;

    private LinkedList _pendingTasks;
    private LinkedList _completedTasks;
    private void drainTaskQueue() {
        for (Iterator it = _pendingTasks.iterator(); it.hasNext();) {
            IGETask task = (IGETask)it.next();
            task.execute();
            _completedTasks.addLast(task);
        }
        _pendingTasks.clear();
    }
```

```
    private void enqueueTask(IGETask task) {
        _pendingTasks.addLast(task);
    }


    PrintApplicator(GE geq, Print p) {
        _old2new_bdmap = new TreeMap();
        _old2new_bsmap = new HashMap();
        _geqOriginal = geq;
        _geqPrinted = geq.duplicate(_old2new_bdmap, _old2new_bsmap);

        _crtr2new_bdmap = new TreeMap();

        _pendingTasks = new LinkedList();
        _completedTasks = new LinkedList();

        apply(p);
    }

    GE getPrinted() {
        return _geqPrinted;
    }

    private void apply(Print p) {

        System.out.println("XYZ XXXXXXXXXX");

        insertNewBoundaries(p);

        LinkedList additionalTasks = null;
        if (isCriticalBoundaryStrictlyBelowRightBoundary()) {
            reduceConstraintsOnCarrier();
        }
        else {
            Base oldCarrier = getOldCarrier();
            additionalTasks = moveBaseToDualPart1(p, oldCarrier, true);
            moveBaseToDualPart2(additionalTasks);
        }

        moveTransportBases(p);

        collapseAlignedVariableBases(p);
        eliminateBasesStartingLeftOfCriticalBoundary(p);
        eliminateUselessBoundaries(p);
    }

    private void eliminateUselessBoundaries(Print p) {
```

```
        Boundary crOld = getOldCriticalBoundary();
        Boundary crNew = Old2NewBoundary(crOld);

        for (Iterator it = _geqPrinted.iteratorBoundaries(); it.hasNext();) {
            Boundary bdNew = (Boundary)it.next();
            if (_geqPrinted.isUseless(bdNew)) {
                enqueueTask(new GETask_DeleteBoundary(bdNew, _geqPrinted));
            }

        }
        drainTaskQueue();
    }

    private void eliminateBasesStartingLeftOfCriticalBoundary(Print p) {
        Boundary crOld = getOldCriticalBoundary();
        Boundary crNew = Old2NewBoundary(crOld);

        for (Iterator it = _geqPrinted.iteratorBases(); it.hasNext();) {
            Base baseNew = (Base)it.next();

            if (baseNew.getBegin().getID() < crNew.getID()) {
                enqueueTask(new GETask_DeleteBase(baseNew, _geqPrinted));
            }
        }
        drainTaskQueue();
    }

    private void collapseAlignedVariableBases(Print p) {
        for (Iterator it = _geqPrinted.iteratorBases(); it.hasNext();) {
            Base baseNew = (Base)it.next();
            if (baseNew.getLabel().isConstant()) continue;

            Base dualNew = baseNew.getDual();

            if (!dualNew.isEmpty() &&
                ((baseNew.getBegin() == dualNew.getBegin()) &&
                 (baseNew.getEnd() == dualNew.getEnd()) &&
                 (baseNew.getLabel() == dualNew.getLabel()))) {
                enqueueTask(new GETask_CollapseBase(baseNew, _geqPrinted));
            }
        }
        drainTaskQueue();
    }

    private void moveTransportBases(Print p) {
        for (Iterator it = _geqOriginal.iteratorBases(); it.hasNext();) {
            Base transportOld = (Base)it.next();
```

```
            if (transportOld == getOldCarrier()) continue;

            if (transportOld.lookupDecorator(makanin.BaseClassDecorator.NAME) instanceof makanin.BaseClassTransport) {
                addConstraintsToCarrier(transportOld);

                Base transportNew = Old2NewBase(transportOld);
                System.out.println("XYZ Moving transport base "+transportNew);

                LinkedList additionalTasks = moveBaseToDualPart1(p, transportOld, false);
                moveBaseToDualPart2(additionalTasks);
            }
        }
        drainTaskQueue();
    }

    private void insertNewBoundaries(Print p) {

        boolean flipped = isCarrierFlipped();

        Boundary startOld = p.getBegin();
        Boundary startNew = Old2NewBoundary(startOld);

        System.out.println("XYZ*** p.getBegin(): "+p.getBegin());
        System.out.println("XYZ*** p.getEnd(): "+p.getEnd());
        System.out.println("XYZ*** startNew: "+startNew);

        for (Iterator it=p.iteratorPrintNodes(!flipped); it.hasNext();) {
            PrintNode pn = (PrintNode)it.next();
            System.out.println("XYZ PrintNode: "+pn);
        }

        for (Iterator it=p.iteratorPrintNodes(!flipped); it.hasNext();) {
            PrintNode pn = (PrintNode)it.next();

            if (pn.getSource() == PrintNode.CARRIER_TR) {
                Boundary equiv = getEquivalentExistingBoundaryNew(p, pn);
                if (equiv == null) {
                    boolean after = true;

                    // System.out.println("add after: "+startNew);

                    GETask_BoundaryInsertion task = new GETask_BoundaryInsertion(startNew, after, _geqPrinted);
                    enqueueTask(task);
                    drainTaskQueue();
                    startNew = task.getNewBoundary();

                    System.out.println("XYZ pn "+pn+" ++> "+startNew);
```

```
                    _crtr2new_bdmap.put(pn.getBoundary(), startNew);
                }
                else {
                    System.out.println("XYZ pn "+pn+" ==> "+equiv);
                    _crtr2new_bdmap.put(pn.getBoundary(), equiv);
                }
            }
            else {
                startNew = Old2NewBoundary(pn.getBoundary());
                // System.out.println("pushed startNew: "+startNew);
            }
        }
    }

    private boolean isEquivalentToExistingBoundary(Print p, PrintNode pntest) {
        return (getEquivalentExistingBoundaryNew(p, pntest) != null);
    }

    private Boundary getEquivalentExistingBoundaryNew(Print p, PrintNode pntest) {
        Boundary beforeBd=isEquivalentToExistingBoundaryNewBefore(p,pntest);
        boolean before = (beforeBd!=null);

        Boundary afterBd=isEquivalentToExistingBoundaryNewAfter(p,pntest);
        boolean after = (afterBd!=null);
        if (before && after && (beforeBd != afterBd)) {
            System.out.println("XYZ testing for equivs to "+pntest);
            System.out.println("XYZ equiv to "+afterBd);
            System.out.println("XYZ equiv to "+beforeBd);
            throw new RuntimeException("PrintApplicator.isEquivalentToExistingBoundary finds something both before and aft
er");
        }
        if (before) {
            return beforeBd;
        }
        if (after) {
            return afterBd;
        }
        return null;
    }

    private Boundary isEquivalentToExistingBoundaryNewBefore(Print p, PrintNode pntest) {
        PrintNode pn = pntest;
        do {
            if (pn!=null) {
                if (pn.getOffset() == PrintNode.NONZERO) {
                    return null;
                }
```

```
                if (pn.getSource() == PrintNode.DUAL) {
                    return Old2NewBoundary(pn.getBoundary());
                }
                else if (pn.getSource() == PrintNode.CARRIER_TR) {
                    Boundary inserted = CarrierTranspose2NewBoundary(pn.getBoundary());
                    if (inserted!=null) {
                        return inserted;
                    }
                }
            }
            pn = p.prevPrintNode(pn);
        }
        while (pn!=null);
        return null;
    }

    private Boundary isEquivalentToExistingBoundaryNewAfter(Print p, PrintNode pntest) {
        PrintNode pn = pntest;
        if (pn.getSource() == PrintNode.DUAL) {
            return Old2NewBoundary(pn.getBoundary());
        }
        else if (pn.getSource() == PrintNode.CARRIER_TR) {
            Boundary inserted = CarrierTranspose2NewBoundary(pn.getBoundary());
            if (inserted!=null) {
                return inserted;
            }
        }

        do {
            pn = p.nextPrintNode(pn);
            if (pn!=null) {
                if (pn.getOffset() == PrintNode.NONZERO) {
                    return null;
                }
                if (pn.getSource() == PrintNode.DUAL) {
                    return Old2NewBoundary(pn.getBoundary());
                }
                else if (pn.getSource() == PrintNode.CARRIER_TR) {
                    Boundary inserted = CarrierTranspose2NewBoundary(pn.getBoundary());
                    if (inserted!=null) {
                        return inserted;
                    }
                }
            }
        }
        while (pn!=null);
```

```
        return null;
    }

    private boolean isCriticalBoundaryStrictlyBelowRightBoundary() {
        boolean answer = true;
        Base carrier = getOldCarrier();
        Boundary cr = getOldCriticalBoundary();
        if (cr.getID() < carrier.getEnd().getID()) {
            answer = true;
        }
        else {
            answer = false;
        }
        return answer;
    }

    private boolean isCarrierFlipped() {

        Base carrierOld = getOldCarrier();
        Base carrierNew = Old2NewBase(carrierOld);
        Base carrierDualNew = carrierNew.getDual();

        if (( carrierNew.getLabel().isPositive() && !carrierDualNew.getLabel().isPositive()) ||
            (!carrierNew.getLabel().isPositive() &&  carrierDualNew.getLabel().isPositive())) {
            return true;
        }
        else return false;
    }

    private LinkedList moveBaseToDualPart1(Print p, Base baseOld, boolean isCarrier) {
        Base dualOld = baseOld.getDual();
        Constraint consOld = baseOld.getConstraint();
        Constraint consdualOld = baseOld.getConstraint();

        Base baseNew = Old2NewBase(baseOld);
        Base dualNew = Old2NewBase(dualOld);
        Constraint consNew = baseNew.getConstraint();
        Constraint consdualNew = dualNew.getConstraint();

        LinkedList additionalTasks = new LinkedList();

        Iterator itOrig=consOld.iteratorBoundary();
        for (Iterator it=consNew.iteratorBoundary(); it.hasNext();) {
            Boundary bdNew = (Boundary)it.next();
            Boundary bdDualNew = consNew.getDual(bdNew);

            if (!isCarrier) {
```

```
                System.out.println("XYZ GETask_ConstraintDeletion base "+baseNew.getLabel()+"@"+bdNew+" XXX "+dualNew.getL
abel()+"@"+bdDualNew);
            }
            enqueueTask(new GETask_ConstraintDeletion(baseNew, bdNew, bdDualNew, _geqPrinted));

            Boundary bdOrig =(Boundary)itOrig.next();

            Boundary bdOrig_tr_New = CarrierTranspose2NewBoundary(bdOrig);

            /*
            PrintNode pOrig = p.get(bdOrig, PrintNode.CARRIER_TR);
            if (pOrig == null) {
                throw new RuntimeException("PrintApplicator.moveBaseToDualPart1: pOrig is null");
            }

            bdOrig_tr_New = getEquivalentExistingBoundaryNew(p, pOrig);
             */

            if (bdOrig_tr_New == null) {
                System.out.println("searching for = "+bdOrig);
                System.out.println("print = "+p);
                throw new RuntimeException("PrintApplicator.moveBaseToDualPart1: bdOrig_tr is null");
            }

            if (!isCarrier) {
                System.out.println("XYZ GETask_ConstraintAddition base "+baseNew.getLabel()+"@"+bdOrig_tr_New+" === "+dual
New.getLabel()+"@"+bdDualNew);
            }

            additionalTasks.addLast(new GETask_ConstraintAddition(baseNew, bdOrig_tr_New, bdDualNew, _geqPrinted));
        }
        drainTaskQueue();

        Boundary begin = CarrierTranspose2NewBoundary(baseOld.getBegin()); // dualNew.getBegin();
        Boundary end = CarrierTranspose2NewBoundary(baseOld.getEnd()); // dualNew.getEnd();
        /*
        if (isCarrierFlipped()) {
            Boundary swap=begin;
            begin=end;
            end=swap;
        }
         */
        additionalTasks.addFirst(new GETask_MoveBase(baseNew, begin, end, _geqPrinted));

        return additionalTasks;
    }
```

```
    private void moveBaseToDualPart2(LinkedList additionalTasks) {
        for (Iterator it=additionalTasks.iterator(); it.hasNext();) {
            IGETask task = (IGETask)it.next();
            enqueueTask(task);
        }
        drainTaskQueue();
    }

    private void reduceConstraintsOnCarrier() {
        Base carrier = Old2NewBase(getOldCarrier());
        Boundary cr = Old2NewBoundary(getOldCriticalBoundary());

        // System.out.println("Carrier is "+carrier);

        Constraint cons = carrier.getConstraint();
        for (Iterator it=cons.iteratorBoundary(); it.hasNext();) {
            Boundary bdi=(Boundary)it.next();
            if (bdi.getID() <  cr.getID()) {
                Boundary bdi_dual = cons.getDual(bdi);
                enqueueTask(new GETask_ConstraintDeletion(carrier, bdi, bdi_dual, _geqPrinted));
            }
        }

        drainTaskQueue();
    }

     private void addConstraintsToCarrier(Base transportOld) {
        Base transportNew = Old2NewBase(transportOld);

        Base carrierNew = Old2NewBase(getOldCarrier());
        Boundary cr = Old2NewBoundary(getOldCriticalBoundary());

        // System.out.println("Carrier is "+carrier);

        Constraint consCarrier = carrierNew.getConstraint();
        Constraint consTransportBase = transportNew.getConstraint();

        for (Iterator it=consTransportBase.iteratorBoundary(); it.hasNext();) {
            Boundary bdtrans=(Boundary)it.next();
            if (cr.getID() < bdtrans.getID()) {
                Boundary bdtrans_dual = consTransportBase.getDual(bdtrans);
                enqueueTask(new GETask_ConstraintDeletion(carrierNew, bdtrans, bdtrans_dual, _geqPrinted));
            }
        }
    }

    private Boundary Old2NewBoundary(Boundary bd) {
```

```
        Boundary bdnew = (Boundary)_old2new_bdmap.get(bd);
        if (bdnew==null) {
            throw new RuntimeException("PrintApplicator.Old2NewBoundary unknown boundary");
        }
        return bdnew;
    }

    private Boundary CarrierTranspose2NewBoundary(Boundary bd) {
        Boundary bdnew = (Boundary)_crtr2new_bdmap.get(bd);
        return bdnew;
    }

    private Base Old2NewBase(Base bs) {
        Base bsnew = (Base)_old2new_bsmap.get(bs);
        if (bsnew==null) {
            throw new RuntimeException("PrintApplicator.Old2NewBase unknown base");
        }
        return bsnew;
    }

    private Base getOldCarrier() {
        Carrier ca = (Carrier)_geqOriginal.lookupDecorator(Carrier.NAME);
        if (ca == null) {
            throw new RuntimeException("PrintApplicator.Main: unknown carrier");
        }
        Base carrier_base = ca.getBase();
        if (carrier_base == null) {
            throw new RuntimeException("PrintApplicator.Main: carrier_base is null");
        }
        return carrier_base;
    }

    private Boundary getOldCriticalBoundary() {
        Base carrier_base = getOldCarrier();
        CriticalBoundary cr = (CriticalBoundary)_geqOriginal.lookupDecorator(CriticalBoundary.NAME);
        if (cr == null) {
            throw new RuntimeException("PrintApplicator.Main: unknown critical boundary");
        }
        Boundary critical_boundary = cr.getBoundary();
        if (critical_boundary == null) {
            throw new RuntimeException("PrintApplicator.Main: critical_boundary is null");
        }
        return critical_boundary;
    }

    private Base getOldCarrierDual() {
        Base carrier_base = getOldCarrier();
```

```
        Base carrier_dual = carrier_base.getDual();
        return carrier_dual;
    }


    public String toString() {
        String s="";
        int i=1;
        for (Iterator it = _completedTasks.iterator(); it.hasNext();) {
            IGETask task = (IGETask)it.next();
            s+="{\\underline{Step "+i+"}:} ";
            s+=task.toString();
            if (it.hasNext()) {
                s+="\\\\\n";
            }
            else {
                s+="\\\\[0.2in]\n";
            }
            i++;
        }
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.GE;
import ge.Base;
import java.util.Iterator;
import utility.CompositeIterator;
import utility.CompositeIterator.State;

/**
 *
 * @author grouptheory
 */
public class PrintIterator implements Iterator {

    private GE _geq;
    private PrintProbe _pp;
    private Iterator _it;
    private Print _nextPrint;

    PrintIterator(GE geq) {
        _geq = geq;
        _pp = new PrintProbe(geq);
        _it = _pp.iteratorTreeNodes();

        _nextPrint = nextPrint();
    }

    public boolean hasNext() {
        return (_nextPrint!=null);
    }

    public Object next() {
        Print nextPrint = _nextPrint;
        _nextPrint = nextPrint();
        return nextPrint;
    }

    public void remove() {
        throw new RuntimeException("PrintIterator.remove: not implemented");
    }

    private Print nextPrint() {
```

```
        Print answer = null;
        while (_it.hasNext()) {
            Print pr = new Print(isCarrierFlipped());
            CompositeIterator.State cs = (CompositeIterator.State)_it.next();

            for (Iterator itsub=cs.iteratorComposableStates(); itsub.hasNext();) {
                PrintNode pn = (PrintNode)itsub.next();
                pr.append(pn);
            }

            if (PrintValidator.validate(pr, _geq)) {
                answer = pr;
                break;
            }
        }
        return answer;
    }


    private Base getCarrier() {
        Carrier ca = (Carrier)_geq.lookupDecorator(Carrier.NAME);
        if (ca == null) {
            throw new RuntimeException("PrintApplicator.Main: unknown carrier");
        }
        Base carrier_base = ca.getBase();
        if (carrier_base == null) {
            throw new RuntimeException("PrintApplicator.Main: carrier_base is null");
        }
        return carrier_base;
    }

    private Base getCarrierDual() {
        Base carrier_base = getCarrier();
        Base carrier_dual = carrier_base.getDual();
        return carrier_dual;
    }

    private boolean isCarrierFlipped() {
        Base carrier = getCarrier();
        Base carrierDual = getCarrierDual();

        if (( carrier.getLabel().isPositive() && !carrierDual.getLabel().isPositive()) ||
            (!carrier.getLabel().isPositive() &&  carrierDual.getLabel().isPositive())) {
            return true;
        }
        else return false;
```

```
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.GE;

/**
 *
 * @author grouptheory
 */
public class PrintIteratorFactory {

    private static PrintIteratorFactory _instance;

    private PrintIteratorFactory() {
    }

    public static PrintIteratorFactory instance() {
        if (_instance == null) {
            _instance = new PrintIteratorFactory();
        }
        return _instance;
    }

    public PrintIterator newPrintIterator(GE geq) {
        return new PrintIterator(geq);
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.Boundary;
import ge.Base;
import ge.GE;
import java.util.LinkedList;
import java.util.Iterator;
import letter.Letter;
import letter.Variable;

/**
 *
 * @author grouptheory
 */
public class PrintNode {

    static class Offset {
        private String _name;
        private Offset(String name) { _name = name; }
        public String toString() { return _name; }
    };

    static Offset ZERO = new Offset("ZERO");
    static Offset NONZERO = new Offset("NONZERO");

    static class Source {
        private String _name;
        private Source(String name) { _name = name; }
        public String toString() { return _name; }
    };

    static Source CARRIER_TR = new Source("CARRIER_TR");
    static Source DUAL = new Source("DUAL");

    private LinkedList _ca_bd;
    private LinkedList _caDual_bd;
    private PrintNode.Offset _offset;
    private Boundary _bd;
    private PrintNode.Source _source;

    PrintNode(GE geq) {
        this();
```

```
        Carrier ca = (Carrier)geq.lookupDecorator(Carrier.NAME);
        if (ca == null) {
            throw new RuntimeException("gutierrez.Main: unknown carrier");
        }
        Base carrier_base = ca.getBase();
        if (carrier_base == null) {
            throw new RuntimeException("gutierrez.Main: carrier_base is null");
        }

        _offset = PrintNode.ZERO;
        _bd = null;
        _source = null;

        for (int i=carrier_base.getBegin().getID(); i<=carrier_base.getEnd().getID(); i++) {
            Boundary bd = geq.getNthBoundary(i);
            _ca_bd.addLast(bd);
        }

        Base carrier_dual = carrier_base.getDual();

        Variable v = (Variable)carrier_base.getLabel();
        Variable vdual = (Variable)carrier_dual.getLabel();

        boolean flipDualBoundaries = false;
        if ((v.isPositive() && !vdual.isPositive()) ||
            (!v.isPositive() && vdual.isPositive())) {
            flipDualBoundaries = true;
        }

        for (int i=carrier_dual.getBegin().getID(); i<=carrier_dual.getEnd().getID(); i++) {
            Boundary bd = geq.getNthBoundary(i);
            if (!flipDualBoundaries) {
                _caDual_bd.addLast(bd);
            }
            else {
                _caDual_bd.addFirst(bd);
            }
            //System.out.print(""+bd+",");
        }
        //System.out.println("");
    }

    PrintNode(LinkedList ca_bd, LinkedList caDual_bd,
              PrintNode.Offset offset, Boundary bd, PrintNode.Source source) {
        this();
        _ca_bd.addAll(ca_bd);
```

```
        _caDual_bd.addAll(caDual_bd);
        _offset = offset;
        _bd = bd;
        _source = source;
    }

    private PrintNode() {
        _ca_bd = new LinkedList();
        _caDual_bd = new LinkedList();
        _offset = PrintNode.ZERO;
        _bd = null;
        _source = null;
    }

    PrintNode.Source getSource() {
        return _source;
    }

    PrintNode.Offset getOffset() {
        return _offset;
    }

    Boundary getBoundary() {
        return _bd;
    }

    PrintNode consumeCarrierBoundary(PrintNode.Offset offset) {
        LinkedList clist = new LinkedList();
        clist.addAll(_ca_bd);
        Boundary bd = (Boundary)clist.getFirst();
        clist.removeFirst();
        return new PrintNode(clist, _caDual_bd, offset, bd, PrintNode.CARRIER_TR);
    }

    PrintNode consumeCarrierDualBoundary(PrintNode.Offset offset) {
        LinkedList clist = new LinkedList();
        clist.addAll(_caDual_bd);
        Boundary bd = (Boundary)clist.getFirst();
        clist.removeFirst();
        return new PrintNode(_ca_bd, clist, offset, bd, PrintNode.DUAL);
    }

    int remainingCarrierBoundaries() {
        return _ca_bd.size();
    }

    int remainingCarrierDualBoundaries() {
```

```
        return _caDual_bd.size();
    }

    public String toString() {
        String s="";

        if (_offset==PrintNode.ZERO) {
            s += "=";
        }
        else if (_offset==PrintNode.NONZERO) {
            s += "<";
        }
        else {
            throw new RuntimeException("PrintNode.toString: unknown offset");
        }

        if (_bd==null) {
            s += "[]";
        }
        else {
            s += _bd;

            if (_source==PrintNode.CARRIER_TR) {
                s += "";
            }
            else if (_source==PrintNode.DUAL) {
                s += "*";
            }
        }

        return s;
    }

    public String toStringLong() {
        String s="";

        s+="Offset: ";
        if (_offset==PrintNode.ZERO) {
            s += "=";
        }
        else if (_offset==PrintNode.NONZERO) {
            s += "<";
        }
        else {
            throw new RuntimeException("PrintNode.toString: unknown offset");
        }
        s+="\n";
```

```
        s+="Boundary: ";
        s += _bd;
        s+="\n";

        s+="Carrier boundaries: ";
        for (Iterator it=_ca_bd.iterator(); it.hasNext();) {
            Boundary bd = (Boundary)it.next();
            s+=bd;
            if (it.hasNext()) s+=",";
        }
        s+="\n";

        s+="Carrier Dual boundaries: ";
        for (Iterator it=_caDual_bd.iterator(); it.hasNext();) {
            Boundary bd = (Boundary)it.next();
            s+=bd;
            if (it.hasNext()) s+=",";
        }
        s+="\n";

        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.GE;
import java.util.Iterator;
import utility.CompositeIterator;

/**
 *
 * @author grouptheory
 */
public class PrintProbe {

    private GE _geq;

    PrintProbe(GE geq) {
        _geq = geq;
    }

    public Iterator iteratorTreeNodes() {
        PrintNode pn = new PrintNode(_geq);
        boolean root = true;
        ComposablePrintNodeIterator cdi = new ComposablePrintNodeIterator(pn, root);
        CompositeIterator compiter = new CompositeIterator(cdi, false);
        return compiter;
    }

    public String toString() {
        String s = "";
        s += "PrintProbe: ";
        return s;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package makanin;

import ge.GE;
import ge.Base;
import ge.Boundary;
import ge.Constraint;
import letter.Letter;
import letter.Variable;
import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
public class PrintValidator {

    static boolean validate(Print p, GE geq) {
        //System.out.println("PrintValidator");

        if ( ! canonicalEqualityOrdering(p,geq)) return false;
        if ( ! carrierExactlyOverlapsDual(p,geq)) return false;
        if ( ! carrierConstraintsOverlapsDualConstraints(p,geq)) return false;
        if ( ! nonCollapsing(p,geq)) return false;
        if ( ! consistentTransport(p,geq)) return false;
        if ( ! consistentConstants(p,geq)) return false;

        return true;
    }

    private static boolean consistentConstants(Print p, GE geq) {
        // constant bases move consistently
        boolean answer = true;

        Boundary lp=p.getBegin();
        Boundary rp=p.getEnd();

        for (Iterator it=geq.iteratorBases(); it.hasNext();) {
            Base bs = (Base)it.next();
            if ( ! bs.getLabel().isConstant()) continue;

            Boundary left=bs.getBegin();
            Boundary right=bs.getEnd();
```

```
            if (left.getID()+1 != right.getID()) {
                throw new RuntimeException("PrintValidator.consistentConstants: constant length > 1");
            }

            if (left.getID()<=lp.getID() && right.getID()<=lp.getID()) {
                // entirely to the left
            }
            else if (left.getID()>=rp.getID() && right.getID()>=rp.getID()) {
                // entirely to right
            }
            else if (left.getID()>=lp.getID() && right.getID()<=rp.getID()) {
                // inside
                int distance = p.compare(left, PrintNode.DUAL, right, PrintNode.DUAL);

                boolean currentanswer = true;
                if (distance != -1) {
                    currentanswer = false;
                }

                if (answer && !currentanswer) {
                    System.out.println("***********************************");
                    System.out.println("consistentConstants rejects a print "+p+"");
                    System.out.println("of ge "+geq+"");
                    System.out.println("because of "+bs+"");
                }

                if (answer && !currentanswer) {
                    System.out.println("consistentConstants ACCEPTS a print "+p+"");
                    System.out.println("of ge "+geq+"");
                    System.out.println("w.r.t. "+bs+"");
                }

                answer = answer && currentanswer;
            }
            else {
                throw new RuntimeException("PrintValidator.consistentConstants: unexpected constant");
            }
        }

        return answer;
    }

    private static boolean consistentTransport(Print p, GE geq) {
        // transport bases move consistently
        boolean answer = true;
        for (Iterator it=geq.iteratorBases(); it.hasNext();) {
```

```
            Base bs = (Base)it.next();
            if (bs.getLabel().isConstant()) continue;

            Base dual = bs.getDual();

            BaseClassDecorator bcd = (BaseClassDecorator)bs.lookupDecorator(BaseClassDecorator.NAME);
            BaseClassDecorator bcdDual = (BaseClassDecorator)dual.lookupDecorator(BaseClassDecorator.NAME);

            if ((bcd instanceof BaseClassTransport) &&
                (bcdDual instanceof BaseClassFixed)) {

                // System.out.println("consistentTransport on "+bs+" whose dual is "+dual);

                Constraint con = bs.getConstraint();
                Constraint con_dual = dual.getConstraint();

                boolean aligned = false;
                for (Iterator it2 = con.iteratorBoundary(); it2.hasNext();) {
                    Boundary bd = (Boundary)it2.next();
                    Boundary bd_dual = con.getDual(bd);

                    if (p.contains(bd_dual, PrintNode.DUAL)) {
                        int b1 = p.compare(bd, PrintNode.CARRIER_TR,
                                           bd_dual, PrintNode.DUAL);

                        if (b1==0) {
                            aligned = true;
                            break;
                        }
                    }
                }

                if (aligned) {
                    boolean strict = true;
                    boolean currentanswer = baseOverlapsDualConstraints(p, geq, bs, !strict);
                    /*
                    if (answer && !currentanswer) {
                        System.out.println("consistentTransport rejects a print "+p+"");
                        System.out.println("of ge "+geq+"");
                        System.out.println("because of "+bs+" whose dual is "+dual);
                    }
                     */
                    answer = answer && currentanswer;
                }

            }
        }
```

```
        return answer;
    }

    private static boolean nonCollapsing(Print p, GE geq) {
        // distinct boundaries go to distinct boundaries
        boolean answer = true;

        // System.out.println("non collapsing: "+p);

        Carrier ca = (Carrier)geq.lookupDecorator(Carrier.NAME);
        if (ca == null) {
            throw new RuntimeException("PrintValidator.nonCollapsing: unknown carrier");
        }
        Base carrier_base = ca.getBase();
        if (carrier_base == null) {
            throw new RuntimeException("PrintValidator.nonCollapsing: carrier_base is null");
        }

        Base carrier_dual = carrier_base.getDual();

        int targetValue = -1;

        int low = carrier_base.getBegin().getID();
        int high = carrier_base.getEnd().getID();
        for (int i=low;i<high;i++) {
            Boundary smaller=geq.getNthBoundary(i);
            Boundary larger=geq.getNthBoundary(i+1);
            int b1 = p.compare(smaller, PrintNode.CARRIER_TR,
                               larger, PrintNode.CARRIER_TR);

            // System.out.println("Comparing "+smaller+":CARRIER with "+larger+":CARRIER");

            if (b1 * targetValue <= 0) answer = false;
        }

        Variable v = (Variable)carrier_base.getLabel();
        Variable vdual = (Variable)carrier_dual.getLabel();

        boolean flipDualBoundaries = false;
        if ((v.isPositive() && !vdual.isPositive()) ||
            (!v.isPositive() && vdual.isPositive())) {
            flipDualBoundaries = true;
        }

        if (flipDualBoundaries) {
            targetValue *= -1;
        }
```

```
        int lowdual = carrier_dual.getBegin().getID();
        int highdual = carrier_dual.getEnd().getID();
        for (int i=lowdual;i<highdual;i++) {
            Boundary smaller=geq.getNthBoundary(i);
            Boundary larger=geq.getNthBoundary(i+1);
            int b2 = p.compare(smaller, PrintNode.DUAL,
                               larger, PrintNode.DUAL);

            // System.out.println("Comparing "+smaller+":DUAL with "+larger+":DUAL");

            if (b2 * targetValue <= 0) answer = false;
        }

        return answer;
    }

    private static boolean carrierConstraintsOverlapsDualConstraints(Print p, GE geq) {
        // the first and last
        boolean answer = true;

        Carrier ca = (Carrier)geq.lookupDecorator(Carrier.NAME);
        if (ca == null) {
            throw new RuntimeException("PrintValidator.carrierConstraintsOverlapsDualConstraints: unknown carrier");
        }
        Base carrier_base = ca.getBase();
        if (carrier_base == null) {
            throw new RuntimeException("PrintValidator.carrierConstraintsOverlapsDualConstraints: carrier_base is null");
        }

        boolean strict = true;
        return baseOverlapsDualConstraints(p, geq, carrier_base, strict);
    }


    private static boolean baseOverlapsDualConstraints(Print p, GE geq, Base base, boolean strict) {
        // the first and last
        boolean answer = true;

        Base dual = base.getDual();

        Variable v = (Variable)base.getLabel();
        Variable vdual = (Variable)dual.getLabel();

        boolean flipDualBoundaries = false;
        if ((v.isPositive() && !vdual.isPositive()) ||
            (!v.isPositive() && vdual.isPositive())) {
```

```
            flipDualBoundaries = true;
        }

        Constraint con_carrier = base.getConstraint();
        Constraint con_dual = dual.getConstraint();

        int a_carrier = 0;
        boolean dual_found1 = false;
        for (Iterator it = con_carrier.iteratorBoundary(); it.hasNext();) {
            Boundary bd = (Boundary)it.next();
            Boundary bd_dual = con_carrier.getDual(bd);

            if (p.contains(bd_dual, PrintNode.DUAL)) {
                dual_found1 = true;
                int b1 = p.compare(bd, PrintNode.CARRIER_TR,
                                   bd_dual, PrintNode.DUAL);

                if (b1!=0) a_carrier=b1;
            }
            else {
                if (strict) {
                    throw new RuntimeException("PrintValidator.carrierConstraintsOverlapsDualConstraints: inconsistent str
ict overlap check 1 fails");
                }
            }
        }

        int a_dual = 0;
        boolean dual_found2 = false;
        for (Iterator it = con_dual.iteratorBoundary(); it.hasNext();) {
            Boundary bd_dual = (Boundary)it.next();
            Boundary bd = con_dual.getDual(bd_dual);

            if (p.contains(bd_dual, PrintNode.DUAL)) {
                dual_found2 = true;

                int b2 = p.compare(bd, PrintNode.CARRIER_TR,
                                   bd_dual, PrintNode.DUAL);

                if (b2!=0) {
                    a_dual=b2;
                }
            }
            else {
                if (strict) {
                    throw new RuntimeException("PrintValidator.carrierConstraintsOverlapsDualConstraints: inconsistent str
ict overlap check 2 fails");
```

```
                }
            }
        }

        if ((a_carrier==0 && a_dual!=0) ||
            (a_carrier!=0 && a_dual==0)) {
            throw new RuntimeException("PrintValidator.carrierConstraintsOverlapsDualConstraints: inconsistent constraints
 between carrier/dual");
        }

        return (a_carrier==0 && (dual_found2 && a_dual==0));
    }

    private static boolean carrierExactlyOverlapsDual(Print p, GE geq) {
        // the first and last boundaries of the carrier and the dual agree
        boolean answer = true;

        Carrier ca = (Carrier)geq.lookupDecorator(Carrier.NAME);
        if (ca == null) {
            throw new RuntimeException("PrintValidator.carrierExactlyOverlapsDual: unknown carrier");
        }
        Base carrier_base = ca.getBase();
        if (carrier_base == null) {
            throw new RuntimeException("PrintValidator.carrierExactlyOverlapsDual: carrier_base is null");
        }

        Base carrier_dual = carrier_base.getDual();

        Variable v = (Variable)carrier_base.getLabel();
        Variable vdual = (Variable)carrier_dual.getLabel();

        boolean flipDualBoundaries = false;
        if ((v.isPositive() && !vdual.isPositive()) ||
            (!v.isPositive() && vdual.isPositive())) {
            flipDualBoundaries = true;
        }

        int b1, b2;
        if (!flipDualBoundaries) {
            b1 = p.compare(carrier_base.getBegin(), PrintNode.CARRIER_TR,
                           carrier_dual.getBegin(), PrintNode.DUAL);
            b2 = p.compare(carrier_base.getEnd(), PrintNode.CARRIER_TR,
                           carrier_dual.getEnd(), PrintNode.DUAL);
            //System.out.println("Comparing "+carrier_base.getBegin()+" & "+carrier_dual.getBegin()+" => "+b1);
            //System.out.println("Comparing "+carrier_base.getEnd()+" & "+carrier_dual.getEnd()+" => "+b2);
        }
        else {
```

```
            b1 = p.compare(carrier_base.getBegin(), PrintNode.CARRIER_TR,
                           carrier_dual.getEnd(), PrintNode.DUAL);
            b2 = p.compare(carrier_base.getEnd(), PrintNode.CARRIER_TR,
                           carrier_dual.getBegin(), PrintNode.DUAL);
            //System.out.println("Comparing "+carrier_base.getBegin()+" & "+carrier_dual.getEnd()+" => "+b1);
            //System.out.println("Comparing "+carrier_base.getEnd()+" & "+carrier_dual.getBegin()+" => "+b2);

        }
        answer = ((b1==0) && (b2==0));

        return answer;
    }

    private static boolean canonicalEqualityOrdering(Print p, GE geq) {
        // all equivalence classes of boundaries in the Print
        // should be ordered by listing CARRIER boundaries
        // before DUAL boundaries
        boolean answer = true;

        PrintNode pn, pnlast;
        pn = pnlast = null;
        for (Iterator it=p.iteratorPrintNodes(true); it.hasNext();) {
            pnlast = pn;
            pn = (PrintNode)it.next();
            if ((pnlast!=null) &&
                (pnlast.getSource()==PrintNode.DUAL) &&
                (pn.getSource()==PrintNode.CARRIER_TR) &&
                (pn.getOffset()==PrintNode.ZERO)) {
                answer = false;
                break;
            }
        }
        return answer;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package params;

/**
 *
 * @author grouptheory
 */
public class MKParams {
    public static final String OUTPUT_DIRECTORY = "output";

    public static final boolean FLAG_REPORT_GE_QUADRATICSYSTEM = true;
    public static final boolean FLAG_REPORT_CANCELLATION_PICTURES = true;
    public static final boolean FLAG_REPORT_GE_STRUCTURES = false;
    public static final boolean FLAG_REPORT_GE_PICTURES = true;
    public static final boolean FLAG_REPORT_MAKANIN_CARRIER = true;
    public static final boolean FLAG_REPORT_MAKANIN_PRINTS = true;
    public static final boolean FLAG_REPORT_DATE = false;
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package report;

import ge.GE;
import ge.GEFactory;
import ge.Latex;
import ge.CancellationDiagramAnalysis_GEDecorator;
import equation.GroupEquation;
import equation.QuadraticSystem;
import cancellation.Diagram;
import cancellation.DiagramTreeNode;
import cancellation.ICancellationDiagramAnalysis;
import cancellation.CancellationDiagramFactory;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
public class CancellationDiagramReport {
    private static String SUFFIX = "-CancellationDiagrams.tex";

    private String _latex;

    String latexHeader() {
        String s = "";
        s+="\\documentclass[final]{article}\n";
        s+="\\usepackage{amssymb,amsmath,amsfonts}\n";
        s+="\\usepackage[dvips]{graphicx}\n";
        s+="\\usepackage{longtable}\n";
        s+="\\usepackage{pstricks-add,pst-slpe}\n";
        s+="\\begin{document}\n";
        if ( ! params.MKParams.FLAG_REPORT_DATE) {
            s+="\\date{}\n";
        }
        return s;
    }

    String latexTitle(GroupEquation eq) {
        String s="";
```

```
        s+="\\title{\n";
        s+="  {\\Large The Cancellation Diagrams of \\\\";
        s += "$";
        s += equation.Latex.instance().renderGroupEquation(eq);
        s += "$";
        s += "\\\\ in a Free Group}\n";
        s+="  {\\normalsize\n";
        s+="   \\author{Bilal Khan\n";
        s+="        \\thanks{Department of Mathematics and Computer Science, John Jay College of Criminal Justice, City Un
iversity of New York (CUNY).}\n";
        s+="   \\and M-K Solver\n";
        s+="        \\thanks{Software developed with support from the National Security Agency Grant H98230-06-1-0042.}\n"
;
        s+="           }\n";
        s+="  }\n";
        s+="}\n\n";
        s+="\\maketitle\n\n";
        return s;
    }

    String latexFooter() {
        String s = "";
        s+="\\end{document}\n";
        return s;
    }

    CancellationDiagramReport(GroupEquation eq) {
        GroupEquation prob = new GroupEquation(eq);

        ICancellationDiagramAnalysis analysis;
        /*
            analysis = CancellationDiagramFactory.instance().newDiagramTree(prob);
        */
        analysis = CancellationDiagramFactory.instance().newDiagramProbe(prob);

        analysis.addDecorator(new CancellationDiagramAnalysis_GEDecorator());

        _latex = "";
        _latex += latexHeader();
        _latex += latexTitle(prob);

        _latex += cancellation.Latex.instance().renderCancellationDiagramAnalysis(analysis);

        _latex +=  latexFooter();
    }

    void save(String filename) {
```

```
        BufferedWriter outfile;
        try {
            outfile = new BufferedWriter(new FileWriter(filename+SUFFIX));
            outfile.write(_latex);
            outfile.close();
        } catch (IOException e) {
            throw new RuntimeException("Report file could not be written to");
        }
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package report;


import ge.GE;
import ge.GEFactory;
import ge.Latex;
import ge.CancellationDiagramAnalysis_GEDecorator;
import equation.GroupEquation;
import equation.QuadraticSystem;
import cancellation.Diagram;
import cancellation.DiagramTreeNode;
import cancellation.ICancellationDiagramAnalysis;
import cancellation.CancellationDiagramFactory;
import cancellation.DiagramDegeneracyTester;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Iterator;
import makanin.CarrierFactory;
import makanin.CriticalBoundaryFactory;
import makanin.BaseClassDecoratorFactory;

/**
 *
 * @author grouptheory
 */
public class GEPrintsReport {
    private static String SUFFIX = "-Prints.tex";

    private String _latex;

    String latexHeader() {
        String s = "";
        s+="\\documentclass[final]{article}\n";
        s+="\\usepackage{amssymb,amsmath,amsfonts}\n";
        s+="\\usepackage[dvips]{graphicx}\n";
        s+="\\usepackage{longtable}\n";
        s+="\\usepackage{verbatim}\n";
        s+="\\usepackage{pstricks-add,pst-slpe}\n";
        s+="\\begin{document}\n";
        if ( ! params.MKParams.FLAG_REPORT_DATE) {
            s+="\\date{}\n";
```

```
        }
        return s;
    }

    String latexTitle(GroupEquation eq) {
        String s="";
        s+="\\title{\n";
        s+="  {\\Large The Prints of the Generalized Equations of \\\\";
        s += "$";
        s += equation.Latex.instance().renderGroupEquation(eq);
        s += "$";
        s += "\\\\ in a Free Group}\n";
        s+="  {\\normalsize\n";
        s+="   \\author{Bilal Khan\n";
        s+="        \\thanks{Department of Mathematics and Computer Science, John Jay College of Criminal Justice, City Un
iversity of New York (CUNY).}\n";
        s+="   \\and M-K Solver\n";
        s+="        \\thanks{Software developed with support from the National Security Agency Grant H98230-06-1-0042.}\n"
;
        s+="           }\n";
        s+="  }\n";
        s+="}\n\n";
        s+="\\maketitle\n\n";
        return s;
    }

    String latexFooter() {
        String s = "";
        s+="\\end{document}\n";
        return s;
    }

    GEPrintsReport(GroupEquation eq) {
        GroupEquation prob = new GroupEquation(eq);

        ICancellationDiagramAnalysis analysis;
        /*
            analysis = CancellationDiagramFactory.instance().newDiagramTree(prob);
        */
            analysis = CancellationDiagramFactory.instance().newDiagramProbe(prob);

        analysis.addDecorator(new CancellationDiagramAnalysis_GEDecorator());

        _latex = "";
        _latex += latexHeader();
        _latex += latexTitle(prob);
```

```
        int i=0;
        for (Iterator it = analysis.iteratorDiagramTreeNodes(); it.hasNext();) {
            DiagramTreeNode dtn = (DiagramTreeNode)it.next();

            _latex += "\\section{Generalized Equation \\#$"+i+"$}\n";

            /*
            if (DiagramDegeneracyTester.isDegenerate(dtn.getDiagram())) {
                continue;
            }

            if (!dtn.getLeaf()) {
                // this can happen if |w| is odd and
                // we return back to 1 with at the
                // penultimate symbol.
                continue;
            }
            */
            QuadraticSystem qs = analysis.getQuadraticSystem();
            Diagram d = dtn.getDiagram();

            GEFactory gef = GEFactory.instance();
            GE geq = gef.newGE(d, qs);

            CarrierFactory.applyToGE(geq);
            CriticalBoundaryFactory.applyToGE(geq);
            BaseClassDecoratorFactory.applyToAllBases(geq);

            if (params.MKParams.FLAG_REPORT_GE_QUADRATICSYSTEM) {
                _latex += equation.Latex.instance().renderQSAsText(qs);
            }
            if (params.MKParams.FLAG_REPORT_GE_PICTURES) {
                _latex += makanin.Latex.instance().renderGEasGraphics(geq);
            }
            if (params.MKParams.FLAG_REPORT_MAKANIN_CARRIER) {
                _latex += makanin.Latex.instance().renderGEasText(geq);
            }
            if (params.MKParams.FLAG_REPORT_MAKANIN_PRINTS) {
                _latex += makanin.Latex.instance().renderPrintsasText(geq);
            }
            i++;
        }

        _latex +=  latexFooter();
    }

    void save(String filename) {
```

```
        BufferedWriter outfile;
        try {
            outfile = new BufferedWriter(new FileWriter(filename+SUFFIX));
            outfile.write(_latex);
            outfile.close();
        } catch (IOException e) {
            throw new RuntimeException("Report file could not be written to");
        }
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package report;

import equation.GroupEquation;

/**
 *
 * @author grouptheory
 */
public class Latex {

    private static Latex _instance;

    private Latex() {
    }

    public static Latex instance() {
        if (_instance == null) {
            _instance = new Latex();
        }
        return _instance;
    }


}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package report;

import equation.GroupEquation;
import cancellation.CancellationDiagramTree;
import cancellation.CancellationDiagramFactory;

/**
 *
 * @author grouptheory
 */
public class Main {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        MetaReport.create("z1-.c1+.z1-.c2-.", "exp1");

    }

}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package report;


import equation.GroupEquation;
import cancellation.CancellationDiagramTree;
import cancellation.CancellationDiagramFactory;

/**
 *
 * @author grouptheory
 */
public class MetaReport {

    public static void create(String equationAsString, String experimentName) {

        GroupEquation problem = new GroupEquation(equationAsString);

        CancellationDiagramReport report = new CancellationDiagramReport(problem);
        report.save(""+params.MKParams.OUTPUT_DIRECTORY+"/"+experimentName);

                    // System.out.println("XXXXXXXXXXXXXXXXXXXXXXXXX");


        GEPrintsReport report2 = new GEPrintsReport(problem);
        report2.save(""+params.MKParams.OUTPUT_DIRECTORY+"/"+experimentName);
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package transformations;

import ge.GE;
import hom.Hom;

/**
 *
 * @author grouptheory
 */
public interface Elem {
    GE apply(GE actedOn);
    Hom getHom();
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package utility;

/**
 *
 * @author grouptheory
 */
public abstract class AbstractComposableIterator implements ComposableIterator {

    private Object _state;

    public Object getState() {
        return _state;
    }

    public void setState(Object o) {
        _state = o;
    }

    private ComposableIterator _p;

    public ComposableIterator getParent() {
        return _p;
    }

    public void setParent(ComposableIterator p) {
        _p = p;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package utility;

import utility.IDecorator;
import utility.IDecorable;
import java.util.HashMap;
import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
public abstract class AbstractDecorable implements IDecorable {
    private HashMap _name2dec;
    private HashMap _dec2name;

    protected AbstractDecorable() {
        _name2dec = new HashMap();
        _dec2name = new HashMap();
    }

    public void attachDecorator(String name, IDecorator dec) {
        if (_name2dec.containsKey(name)) {
            throw new RuntimeException("AbstractDecorable.attachDecorator: _name2dec duplicate key: "+name);
        }
        if (_name2dec.containsValue(dec)) {
            throw new RuntimeException("AbstractDecorable.attachDecorator: _name2dec duplicate val");
        }
        if (_dec2name.containsKey(dec)) {
            throw new RuntimeException("AbstractDecorable.attachDecorator: _dec2name duplicate key");
        }
        if (_dec2name.containsValue(name)) {
            throw new RuntimeException("AbstractDecorable.attachDecorator: _dec2name duplicate val: "+name);
        }

        // Thread.dumpStack();

        _name2dec.put(name, dec);
        _dec2name.put(dec, name);
    }

    public void detachDecorator(IDecorator dec) {
        String name = lookupDecoratorName(dec);
```

```
        if (!_name2dec.containsKey(name)) {
            throw new RuntimeException("AbstractDecorable.detachDecorator: _name2dec unknown key");
        }
        if (!_dec2name.containsKey(dec)) {
            throw new RuntimeException("AbstractDecorable.detachDecorator: _dec2name unknown key");
        }

        _dec2name.remove(dec);
        _name2dec.remove(name);
    }

    public String lookupDecoratorName(IDecorator dec) {
        if (!_dec2name.containsKey(dec)) {
            throw new RuntimeException("AbstractDecorable.lookupDecoratorName: _dec2name unknown key");
        }
        return (String)_dec2name.get(dec);
    }

    public IDecorator lookupDecorator(String name) {
        if (!_name2dec.containsKey(name)) {
            throw new RuntimeException("AbstractDecorable.lookupDecoratorName: _name2dec unknown key");
        }
        return (IDecorator)_name2dec.get(name);
    }

    public Iterator iteratorDecorators() {
        return _name2dec.values().iterator();
    }

}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package utility;

import utility.IDecorator;
import utility.IDecorable;

/**
 *
 * @author grouptheory
 */
public abstract class AbstractDecorator implements IDecorator {

    private IDecorable _owner;

    protected AbstractDecorator() {
    }

    public void setOwner(IDecorable owner) {
        if ((_owner != null) && (_owner != owner)) {
            throw new RuntimeException("BaseDecorator.attach: already attached");
        }

        if (_owner != owner) {
            _owner = owner;
        }
    }

    public IDecorable getOwner() {
        return _owner;
    }
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package utility;

import java.util.Iterator;

/**
 *
 * @author grouptheory
 */
public interface ComposableIterator extends Iterator {

    ComposableIterator newComposableIterator(ComposableIterator parent);

    Object getState();
    void setState(Object o);

    ComposableIterator getParent();
    void setParent(ComposableIterator p);
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package utility;

import java.util.Iterator;
import java.util.HashMap;
import java.util.LinkedList;


/**
 *
 * @author grouptheory
 */
public class CompositeIterator implements Iterator {

    private LinkedList _subIterators;
    private CompositeIterator.State _next;
    private boolean _leafOnly = false;

    public CompositeIterator(ComposableIterator root, boolean leafOnly) {
        _subIterators = new LinkedList();
        root.setParent(null);
        _subIterators.addLast(root);
        _next = computeNext();
        _leafOnly = leafOnly;
    }

    public boolean hasNext() {
        return (_next!=null);
    }

    public Object next() {
        CompositeIterator.State answerFull = _next;
        // System.out.println("DEBUG pre "+answer.getLeafIteratorState());
        _next = computeNext();
        // System.out.println("DEBUG post "+answer);
        if (_subIterators.size() == 0) {
            _next = null;
        }

        Object answer = answerFull;
        if (_leafOnly) {
            answer = answerFull.getLeafIteratorState();
        }
```

```
        return answer;
    }

    public void remove() {
        throw new RuntimeException("CompositeIterator.remove: not implemented");
    }

    private CompositeIterator.State computeNext() {
        ComposableIterator subitWithStuff = null;

        for (Iterator it=_subIterators.descendingIterator();it.hasNext();) {
            ComposableIterator subit = (ComposableIterator)it.next();
            if ( ! subit.hasNext()) {
                it.remove();
            }
            else {
                Object subobj = subit.next();
                if (subobj == null) {
                    it.remove();
                }
                else {
                    subit.setState(subobj);
                    subitWithStuff = subit;
                    break;
                }
            }
        }

        while (subitWithStuff != null) {
            ComposableIterator nextIter = subitWithStuff.newComposableIterator(subitWithStuff);
            if (nextIter != null) {
                if (nextIter.hasNext()) {
                    Object subobj = nextIter.next();
                    if (subobj!=null) {
                        nextIter.setState(subobj);
                        nextIter.setParent(subitWithStuff);
                        _subIterators.addLast(nextIter);
                    }
                    else nextIter=null;
                }
                else nextIter=null;
            }

            subitWithStuff = nextIter;
        }

        return new State(this);
```

```
    }

    public String toString() {
        String s="";
        s += "CompositeIterator BEGIN\n";
        s += "levels = "+_subIterators.size()+"\n";
        int ct=0;
        for (Iterator it=_subIterators.iterator();it.hasNext();) {
            ComposableIterator subit = (ComposableIterator)it.next();
            s += subit.toString();
            if (it.hasNext()) s += ",";
            ct++;
        }

        if (_next == null) {
            s += " CompositeIterator.State = null";
        }
        else {
            s += " State="+_next.toString();
        }
        s += "CompositeIterator END\n";
        return s;
    }

    public static class State {
        private LinkedList _subObjects;
        private CompositeIterator _owner;

        State(CompositeIterator owner) {
            _owner = owner;
            _subObjects = new LinkedList();
            for (Iterator it=_owner._subIterators.iterator();it.hasNext();) {
                ComposableIterator subit = (ComposableIterator)it.next();
                Object subobj = subit.getState();
                _subObjects.addLast(subobj);
            }
        }

        public Object getLeafIteratorState() {
            return _subObjects.getLast();
        }

        public Iterator iteratorComposableStates() {
            return _subObjects.iterator();
        }

        public String toString() {
```

```
            String s="";
            s+="CompositeIterator.State BEGIN\n";
            for (Iterator it=_subObjects.iterator();it.hasNext();) {
                Object subobj = it.next();
                s += subobj;
                if (it.hasNext()) s += ",";
            }
            s+="CompositeIterator.State END\n";
            return s;
        }
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package utility;

/**
 *
 * @author grouptheory
 */
public interface IDecorable {
    void attachDecorator(String name, IDecorator dec);
    void detachDecorator(IDecorator dec);

    String lookupDecoratorName(IDecorator dec);
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package utility;

import utility.IDecorable;

/**
 *
 * @author grouptheory
 */
public interface IDecorator {
    void setOwner(IDecorable owner);
}
```

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package utility;

/**
 *
 * @author grouptheory
 */
public class IntegerComposableIterator
        extends AbstractComposableIterator
        implements ComposableIterator {

    private int _base;
    private int _depth;
    private int _value;

    IntegerComposableIterator(int base, int depth) {
        _base = base;
        _depth = depth;
        _value = -1;

        setState(null);
    }

    public ComposableIterator newComposableIterator(ComposableIterator parent) {
        setParent(parent);
        if (_depth > 1) {
            return new IntegerComposableIterator(_base, _depth-1);
        }
        else {
            return null;
        }
    }

    public boolean hasNext() {
        if (_value < _base-1) return true;
        else return false;
    }

    public Object next() {
        _value++;
        return new Integer(_value);
    }
```

```
    public void remove() {
        throw new RuntimeException("CompositeIterator.remove: not implemented");
    }


    public String toString() {
        String s="";
        s += " ICI("+_value+")";
        return s;
    }
}
```

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package utility;

/**
 *
 * @author grouptheory
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

        IntegerComposableIterator ici = new IntegerComposableIterator(2, 6);
        CompositeIterator compiter = new CompositeIterator(ici, false);
        // System.out.println("INITIAL compiter="+compiter.toString());

        int i=1;
        while (compiter.hasNext()) {
            // System.out.println("LOOP-preNEXT "+i+" compiter="+compiter.toString());
            Object o = compiter.next();
            // System.out.println("LOOP-postNEXT "+i+" compiter="+compiter.toString());
            System.out.println("STATE: "+o.toString());
            i++;
        }

        // System.out.println("FINAL compiter="+compiter.toString());


        IntegerComposableIterator ici2 = new IntegerComposableIterator(6,2);
        CompositeIterator compiter2 = new CompositeIterator(ici2, false);
        // System.out.println("INITIAL compiter="+compiter.toString());

        int i2=1;
        while (compiter2.hasNext()) {
            // System.out.println("LOOP-preNEXT "+i+" compiter="+compiter.toString());
            Object o2 = compiter2.next();
            // System.out.println("LOOP-postNEXT "+i+" compiter="+compiter.toString());
            System.out.println("STATE: "+o2.toString());
            i2++;
        }
```

```
        // System.out.println("FINAL compiter="+compiter.toString());
    }
}
```